

УЧИТЬСЯ СТАЛО ПРОЩЕ!



2-е издание

Алгоритмы

для

чайников



Управляй алгоритмами
с помощью данных

—
Создавай собственные
алгоритмы на Python

—
Используй алгоритмы
в реальном мире

Джон Пол Мюллер

Лука Массарон

для
чайников



Алгоритмы для чайников

2-е издание

Джон Пол Мюллер
Лука Массарон

 **БОМБОРА**
ИЗДАТЕЛЬСТВО
Москва

УДК 004.42
ББК 32.973.26-018
М98

John Paul Mueller, Luca Massaron
ALGORITHMS FOR DUMMIES
2nd edition

For Dummies® trademark is the exclusive property of Wiley and is used under license.
Original English language edition Copyright © 2022 by John Wiley & Sons, Inc.
All rights reserved including the right of reproduction in whole in part in any form. This translation published
by arrangement with John Wiley & Sons, Inc.
Алгоритмы для чайников © 2022 by John Wiley & Sons, Inc. Все права, включая право
на воспроизведение, защищены полностью или частично в любой форме.
Этот перевод опубликован по соглашению с John Wiley & Sons, Inc.
Wiley, the Wiley Publishing Logo, For Dummies, Dummies Man and related trade dress are trademarks
or registered trademarks of John Wiley and Sons, Inc. and/or its affiliates in the United States
and/or other countries. Used by permission.
Wiley, логотип the Wiley Publishing, For Dummies, «Для чайников», Dummies Man и связанные с ними
элементы являются зарегистрированными товарными знаками John Wiley and Sons, Inc. и/или
ее филиалами в США и других странах. Используется с разрешения.

Мюллер, Джон Пол.

М98 Алгоритмы для чайников / Джон Пол Мюллер, Лука Массарон ; [перевод с английского А. А. Тарасовой]. — 2-е издание. — Москва : Эксмо, 2025. — 480 с. — (Для чайников).

ISBN 978-5-04-201470-3

Книга «Алгоритмы для чайников» — это простое и понятное руководство по основам алгоритмов и их практическому применению. Вы узнаете, как работают алгоритмы, как их создавать с помощью самого популярного языка программирования Python и как они используются в реальной жизни — от соцсетей до финансовых расчетов. В книге вы найдете наглядные примеры, графики и код. Идеально подходит для начинающих программистов и всех, кто хочет разобраться в основах алгоритмов.

УДК 004.42
ББК 32.973.26-018

ISBN 978-5-04-201470-3

© Тарасова А. А., перевод на русский язык, 2025
© Оформление. ООО «Издательство «Эксмо», 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Об этой книге	14
Глупые предположения	16
Значки, используемые в книге	17
Помимо книги	18
Куда двигаться дальше	19

РАЗДЕЛ 1. НАЧАЛО РАБОТЫ С АЛГОРИТМАМИ..... 21

Глава 1.

Знакомство с алгоритмами..... 23

§ 1. Описание алгоритмов	24
Правильный способ приготовления тостов: определение применения алгоритмов.....	26
Алгоритмы повсюду	29
§ 2. Использование компьютеров для решения проблем.....	30
Максимальное использование возможностей современных ЦП и графических процессоров.....	31
Работа со специализированными чипами	32
Сети: совместное использование – это больше чем забота	33
Использование доступных данных	34
§ 3. Различение проблем и решений.....	35
Корректность и эффективность.....	35
Понимание, что бесплатный сыр только в мышеловке	36
Адаптация стратегии к проблеме	36
Описание алгоритмов на универсальном языке	37
Решение проблем, похожих на кирпичные стены, только сложнее	37
§ 4. Структурирование данных для получения решения	38
Понимание точки зрения компьютера.....	38
Упорядочивание данных имеет значение	39

ГЛАВА 2.

Рассматриваем проектирование алгоритмов..... 40

§ 1. Начинаем решать проблему	41
Моделирование реальных задач.....	42
Поиск решений и контрпримеров.....	44
Стоя на плечах гигантов	45

§ 2. Разделяем и властвуем	46
Избегаем решений методом полного перебора	47
Принцип KISS (Keeping it simple, silly, или «не усложняй») ..	48
Разделение проблемы обычно эффективнее	48
§ 3. Узнаем, что жадность может быть полезной.	49
Применение жадного подхода	49
Поиск хорошего решения	50
§ 4. Вычисление затрат и последующие эвристики.	51
Представление задачи в виде пространства	52
Случайный поиск и везение	53
Использование эвристики и функции стоимости	54
§ 5. Оценка алгоритмов.	55
Моделирование с использованием абстрактных машин	56
Еще более абстрактный подход	57
Работа с функциями	58

ГЛАВА 3.

Работа с Google Colab	62
§ 1. Определение Google Colab	63
Понимание функций Google Colab.	63
§ 2. Работа с блокнотами	69
Создание нового блокнота.	69
Открытие существующих блокнотов.	70
Сохранение блокнотов	72
§ 3. Выполнение общих задач.	74
Создание ячеек с кодом	75
Создание текстовых ячеек	77
Создание специальных ячеек	77
Редактирование ячеек	78
Перемещение ячеек	78
§ 4. Использование аппаратного ускорения	79
§ 5. Выполнение кода	79
§ 6. Получение помощи.	80

ГЛАВА 4.

Выполнение базовых операций с данными на Python	82
§ 1. Выполнение вычислений с использованием векторов и матриц.	83
Понимание скалярных и векторных операций	84
Выполнение умножения векторов	86
Создание матрицы – правильный способ начать	86
Умножение матриц.	87
Определение расширенных матричных операций	89
§ 2. Создание комбинаций правильным способом.	91
Работа с перестановками	91
Перемешивание комбинаций	93

Борьба с повторениями	94
§ 3. Получение желаемых результатов с помощью рекурсии.....	94
Объяснение рекурсии	95
Устранение хвостовой рекурсии.....	98
§ 4. Более быстрое выполнение задач.....	99
Рассматриваем метод «разделяй и властвуй».....	99
Различение разных возможных решений.....	102
ГЛАВА 5.	
Разработка класса для матричных вычислений.....	104
§ 1. Избегание использования NumPy	105
§ 2. Почему стоит использовать классы	107
§ 3. Создание базового класса.....	108
Создание матрицы.....	109
Вывод результирующей матрицы	110
Доступ к конкретным элементам матрицы.....	111
Выполнение скалярного сложения и сложения	
матриц	112
Выполнение умножения.....	113
§ 4. Манипулирование матрицей	117
Транспонирование матрицы.....	117
Вычисление определителя.....	118
Уплощение матрицы (преобразование в одномерный	
список)	122
 РАЗДЕЛ 2. ПОНИМАНИЕ НЕОБХОДИМОСТИ	
СОРТИРОВКИ И ПОИСКА	123
ГЛАВА 6.	
Структурирование данных.....	125
§ 1. Определение необходимости структуры.....	126
Облегчаем понимание содержимого.....	126
Сопоставление данных из разных источников.....	127
Учет необходимости исправления данных.....	129
§ 2. Упорядочивание и нагромождение данных.....	132
Упорядочивание в стеках	132
Использование очередей	134
Поиск данных с помощью словарей	135
§ 3. Работа с деревьями	136
Понимание основ деревьев	137
Построение дерева.....	138
§ 4. Представление отношений в виде графа	140
Выходим за рамки деревьев.....	140
Построение графов	141

ГЛАВА 7.

Упорядочивание и поиск данных	143
§ 1. Сортировка данных с помощью сортировки слиянием и быстрой сортировки.	144
Почему важна сортировка данных.	144
Применение улучшенных методов сортировки	148
§ 2. Использование деревьев поиска и кучи	154
Рассматриваем необходимость эффективного поиска	154
Построение бинарного дерева поиска.	156
Выполнение специализированного поиска с помощью бинарной кучи	158
§ 3. Опираясь на хеширование	159
Раскладываем все по корзинам	160
Избегание коллизий	161
Создание собственной хеш-функции.	163

РАЗДЕЛ 3. ИССЛЕДОВАНИЕ МИРА ГРАФОВ

167

ГЛАВА 8.

Основы теории графов	169
§ 1. Объяснение важности сетей	170
Рассматриваем суть графа.	171
Графы повсюду	173
Социальная сторона графов	175
Понимание подграфов.	176
§ 2. Определение того, как рисовать граф	177
Различаем ключевые атрибуты.	177
Визуализация графа	179
§ 3. Измерение функциональности графа	180
Подсчет ребер и вершин	181
Вычисление центральности	183
§ 4. Перевод графа в числовой формат.	186
Преобразование графа в матрицу	186
Использование разреженных представлений	188
Использование списка для хранения графа.	188

ГЛАВА 9.

Восстанавливая связи.	190
§ 1. Эффективное прохождение графа	191
Создание графа	192
Применение поиска в ширину	193
Применение поиска в глубину	195
Выбор подходящего метода	197

§ 2. Сортировка элементов графа	197
Работа с направленными ациклическими графами (DAG)	198
Использование топологической сортировки	199
§ 3. Сведение к минимальному остовному дереву	200
Исторический контекст поиска минимального остовного дерева	200
Работа с невзвешенными и взвешенными графами	201
Создание примера минимального остовного дерева	201
Выбор правильных алгоритмов	203
Знакомство с приоритетными очередями	204
Использование алгоритма Прима	205
Тестирование алгоритма Краскала	207
Определение наилучшего алгоритма	209
§ 4. Поиск кратчайшего маршрута	210
Определение понятия поиска кратчайшего пути	211
Добавление отрицательного ребра	213
Объяснение алгоритма Дейкстры	215
Объяснение алгоритма Беллмана – Форда	218
Объяснение алгоритма Флойда – Уоршелла	221
ГЛАВА 10.	
Раскрывая секреты графов	226
§ 1. Представление социальных сетей как графов	227
Кластеризация сетей на группы	227
Обнаружение сообществ	230
§ 2. Навигация графа	233
Подсчет степеней разделения	233
Случайное блуждание по графу	235
ГЛАВА 11.	
Поиск нужной веб-страницы	237
§ 1. Поиск мира в поисковой системе	238
Поиск данных в интернете	238
Как найти нужные данные	239
§ 2. Объяснение алгоритма PageRank	240
Понимание логики алгоритма PageRank	241
Объясняя принципы работы PageRank	243
§ 3. Реализация PageRank	243
Реализация скрипта на Python	244
Борьба с наивной реализацией	247
Внедрение скуки и телепортации	251
Заглядывая внутрь поисковой системы	252
Другие применения PageRank	253
§ 4. Выход за рамки парадигмы PageRank	253
Знакомство с семантическими запросами	254
Использование ИИ для ранжирования результатов поиска	254

РАЗДЕЛ 4. ОБРАБОТКА БОЛЬШИХ ДАННЫХ255

ГЛАВА 12.

Управление большими данными257

- § 1. Преобразование энергии в данные258
 - Понимание следствий закона Мура259
 - Поиск данных повсюду261
 - Внедрение алгоритмов в бизнес264
- § 2. Поточковые потоки данных266
 - Анализ потоков с правильным рецептом268
 - Сохраняя нужные данные270
- § 3. Скетчинг ответа на основе потоковых данных274
 - Фильтрация элементов потока по существу275
 - Демонстрация фильтра Блума278
 - Определение количества уникальных элементов281
 - Учимся подсчитывать объекты в потоке283

ГЛАВА 13.

Распараллеливание операций285

- § 1. Управление огромными объемами данных286
 - Понимание параллельной парадигмы287
 - Распределение файлов и операций289
 - Применение решения MapReduce292
- § 2. Разработка алгоритмов для MapReduce297
 - Настройка симуляции MapReduce298
 - Исследование с помощью отображения300

ГЛАВА 14.

Сжатие и сокрытие данных304

- § 1. Уменьшение объема данных305
 - Понимание кодирования305
 - Рассматриваем эффекты сжатия307
 - Выбор конкретного типа сжатия309
 - Выбор правильного кодирования311
 - Кодирование с использованием сжатия Хаффмана314
 - Запоминание последовательностей с помощью LZW317
- § 2. Сокрытие ваших секретов с помощью криптографии321
 - Подстановка символов322
 - Работа с шифрованием AES324

РАЗДЕЛ 5. РЕШЕНИЕ СЛОЖНЫХ ЗАДАЧ327

ГЛАВА 15.

Работа с жадными алгоритмами329

§ 1. Решаем, когда лучше быть жадным	330
Почему жадный подход эффективен	332
Держим жадные алгоритмы под контролем	333
Рассмотрение NP-полных задач	335
§ 2. Как жадность может быть полезна	338
Организация кешированных данных компьютера	338
Конкуренция за ресурсы	340
Возвращаясь к кодированию Хаффмана	343
ГЛАВА 16.	
Применение динамического программирования	347
§ 1. Объяснение динамического программирования	348
Исторический экскурс	349
Как сделать задачи динамическими	350
Преобразование рекурсии в динамическое решение	351
Использование мемоизации	355
§ 2. Открываем лучшие динамичные рецепты	358
Заглядываем в рюкзак	358
Путешествие по городам	363
Приближенный поиск строк	368
ГЛАВА 17.	
Использование рандомизированных алгоритмов	372
§ 1. Определение принципа работы рандомизации	373
Рассмотрим, почему нужна рандомизация	374
Понимание работы вероятности	375
Понимание распределений	377
Моделирование использования метода Монте-Карло	381
§ 2. Внесение случайности в вашу логику	383
Вычисление медианы с помощью быстрого выбора	384
Моделирование методом Монте-Карло	387
Более быстрая сортировка с помощью быстрой сортировки	390
ГЛАВА 18.	
Выполнение локального поиска	392
§ 1. Понимание локального поиска	393
Знакомство с окрестностью	394
§ 2. Представляем приемы локального поиска	396
Объяснение алгоритма восхождения к вершине на примере задачи о ферзях	398
Знакомство с имитацией отжига	401
Избегание повторов с помощью поиска с запретами	403
§ 3. Решение выполнимости булевых схем	404
Решение 2-SAT с помощью рандомизации	405

Реализация кода на Python	406
Осознание важности начальной точки	410
ГЛАВА 19.	
Применение линейного программирования	412
§ 1. Использование линейных функций в качестве инструмента. .	413
Осваиваем необходимую математику	415
Учимся упрощать при планировании	417
Понимание ограничений	420
§ 2. Использование линейного программирования на практике .	421
Настройка PuLP дома	422
Оптимизация производства и выручки	422
ГЛАВА 20.	
Рассмотрение эвристических методов	427
§ 1. Дифференциальная эвристика	428
Рассматривая цели эвристик	429
§ 2. Маршрутизация роботов с использованием эвристики	432
Разведка на неизвестной территории	433
Использование мер расстояния в качестве эвристики.	435
§ 3. Объяснение алгоритмов поиска пути	436
Создание лабиринта.	437
В поисках быстрого маршрута по первому наилучшему совпадению	440
Эвристический обход с помощью A*	444
РАЗДЕЛ 6. ЧАСТЬ ДЕСЯТИ	
449	
ГЛАВА 21.	
Десять алгоритмов, которые меняют мир	451
§ 1. Использование процедур сортировки	452
§ 2. Поиск вещей с помощью процедур поиска	453
§ 3. Встряхиваем вещи с помощью случайных чисел	453
§ 4. Выполнение сжатия данных	454
§ 5. Сохранение данных в секрете	455
§ 6. Изменение домена данных	456
§ 7. Анализ ссылок	456
§ 8. Выявление закономерностей в данных	457
§ 9. Работа с автоматизацией и автоматическими ответами	458
§ 10. Создание уникальных идентификаторов	459
ГЛАВА 22.	
Десять нерешенных алгоритмических проблем	460
§ 1. Быстрое решение проблем	461

§ 2. Более эффективное решение задач 3SUM	461
§ 3. Ускорение умножения матриц	462
§ 4. Определение остановки программы	463
§ 5. Создание и использование односторонних функций	463
§ 6. Умножение действительно больших чисел	464
§ 7. Разделение ресурсов поровну	465
§ 8. Сокращение времени расчета расстояния редактирования	465
§ 9. Игра в игру «Паритет»	466
§ 10. Понимание пространственных проблем	466
ОБ АВТОРАХ	468
Посвящение Джона	468
Посвящение Луки	469
Благодарности Джона	469
Благодарности Луки	470
Предметный указатель	471

ВВЕДЕНИЕ

Вы хотите изучить алгоритмы для учебы или работы. Однако все книги по этой теме, которые вы пробовали читать, оказались скорее хорошим снотворным, чем учебными пособиями. Даже если вам удастся разобраться в загадочных символах, явно написанных сумасшедшим двухлетним ребенком с пристрастием к закорючкам, вы все равно не поймете, зачем вообще нужно что-то знать об алгоритмах. Большинство математических учебников скучны! Однако «Алгоритмы для чайников», 2-е издание, — совсем другое дело. Первое, что вы заметите, — в книге практически нет странных символов (особенно в виде закорючек). Да, некоторые символы встречаются (все-таки это математическая книга), но вместо них вы найдете четкие инструкции по использованию алгоритмов, у которых есть имена и история и которые выполняют полезные задачи. Вы познакомитесь с простыми методами программирования для выполнения удивительных задач, которые заинтригуют ваших друзей. Вы определенно сможете вызвать у них зависть, выполняя математические трюки, которые им не под силу понять. И все это без малейшего напряжения мозга — вы даже не заснете (если только сами этого не захотите). В этом издании книги содержится больше подробностей о том, как работают алгоритмы, и вы даже сможете создать собственный базовый математический пакет, чтобы быть готовым к следующему собеседованию.

Об этой книге

«Алгоритмы для чайников», 2-е издание, — это та книга по математике, которую вы хотели получить в университете, но не получили. Вы узнаете, например, что алгоритмы не являются чем-то новым. В конце концов, вавилоняне использовали алгоритмы для выполнения простых задач еще в 1600 году до нашей эры. Если уж вавилоняне смогли во всем этом разобраться, то вы точно сможете!

В книге есть три особенности, которых вы не найдете в большинстве учебников по математике:

- » Алгоритмы, у которых есть реальные названия и историческая основа, благодаря чему вы сможете запомнить алгоритм и понять, почему кто-то потратил время на его создание.

- » Простые объяснения того, как алгоритм выполняет впечатляющие операции по обработке данных, анализу информации или прогнозированию вероятностей.
- » Код, который показывает, как использовать алгоритм, не погружаясь в загадочные символы, понятные только людям с математическим образованием.

Одна из особенностей книги — акцент на использовании правильных инструментов. В книге для выполнения различных задач используется Python. Python обладает особыми возможностями, которые значительно упрощают работу с алгоритмами. Например, Python предоставляет доступ к огромному набору пакетов, позволяющих делать практически все, что можно себе представить, и даже больше. Однако, в отличие от многих текстов, использующих Python, эта книга не перегружает вас пакетами. Мы используем избранную группу пакетов, которые обеспечивают большую гибкость с широкой функциональностью, но при этом не требуют от вас никаких финансовых затрат. Вы можете пройти всю эту книгу, не потратив ни копейки своих кровно заработанных денег.

В книге вы также откроете для себя некоторые интересные методы. Самое важное — вы не просто увидите, как алгоритмы используются для выполнения задач, но и получите объяснение того, как они работают. В отличие от многих других книг, «Алгоритмы для чайников», 2-е издание, позволяют вам полностью понять, что вы делаете, но без необходимости иметь докторскую степень по математике. Каждый из примеров показывает ожидаемый результат и объясняет, почему этот результат важен. У вас не останется ощущения, что чего-то не хватает.

Конечно, вы все еще можете беспокоиться о среде программирования, но и здесь книга не оставит вас в неведении. Она опирается на Google Colab как среду программирования (хотя вы также можете легко использовать Jupyter Notebook). Поскольку доступ к Colab осуществляется через браузер, вы можете программировать где угодно и когда угодно — даже на смартфоне в кабинете стоматолога или, возможно, стоя на голове во время просмотра повторов любимого сериала.

Чтобы помочь вам усвоить материал, в книге используются следующие условные обозначения:

- » Текст, который нужно набрать в точности, как он приведен в книге, выделен **полужирным** шрифтом. Исключение составляют пошаговые инструкции: поскольку каждый шаг выделен полужирным шрифтом, текст для набора в них не выделяется.

- » Слова, которые нужно ввести и которые выделены *курсивом*, используются как заполнители, что означает необходимость заменить их чем-то подходящим для вас. Например, если вы видите: «Введите *Ваше имя* и нажмите Enter», – вам нужно заменить *Ваше имя* своим настоящим именем.
- » Мы также используем *курсив* для терминов, которым даем определение. Это значит, что вам не придется обращаться к другим источникам за нужными определениями.
- » Веб-адреса и программный код набраны **моноширинным** шрифтом. Если читаете цифровую версию этой книги на устройстве, подключенном к интернету, вы можете перейти по активной ссылке, чтобы посетить соответствующий веб-сайт, например: <http://www.dummies.com>.
- » Когда вам нужно выполнить последовательность команд, вы увидите их разделенными специальной стрелкой, например: File ⇨ New File («Файл ⇨ Новый файл»), что означает необходимость нажать сначала «Файл», а затем «Новый файл».

Глупые предположения

Вам может показаться странным, что мы сделали какие-то предположения о вас, а ведь мы даже еще не встречались! Хотя большинство предположений действительно неразумны, но мы сделали определенные допущения, чтобы обозначить отправную точку книги.

Первое предположение заключается в том, что вы знакомы с платформой, которую хотите использовать, поскольку книга не дает никаких рекомендаций в этом отношении. (Впрочем, в главе 3 рассказывается, как получить доступ к Google Colab через браузер и использовать его для работы с примерами кода из книги.) Чтобы предоставить вам максимум информации о Python в контексте алгоритмов, книга не рассматривает никаких вопросов, специфичных для конкретных платформ. Вам действительно нужно знать, как устанавливать приложения, пользоваться ими и в целом работать с выбранной платформой, прежде чем начинать работу с книгой.

Книга не является учебником по математике. Да, вы увидите много примеров сложных математических вычислений, но основной акцент

сделан на том, чтобы помочь вам использовать Python для выполнения типовых задач с помощью алгоритмов, а не на изучении математической теории. Тем не менее вы получите объяснения многих алгоритмов, используемых в книге, чтобы понять принцип их работы. Главы 1 и 2 проведут вас через все необходимое для успешного использования книги. Глава 5 — особенная: в ней рассказывается о том, как создать собственную математическую библиотеку, что существенно поможет вам понять, как математика работает с кодом при создании многократно используемого пакета. К тому же упоминание в резюме о том, что вы создали собственную математическую библиотеку, будет смотреться весьма эффектно.

Еще предполагается, что у вас есть доступ к интернету. По всей книге разбросаны многочисленные ссылки на онлайн-материалы, которые обогатят ваш учебный опыт. Однако дополнительные источники принесут пользу, только если вы действительно найдете их и воспользуетесь ими. Доступ к интернету вам необходим и для работы с Google Colab.

Значки, используемые в книге

Читая книгу, вы встретите на полях значки, указывающие на материал, заслуживающий внимания (или нет, в зависимости от случая). Вот что означают эти значки:

 СОВЕТ	Советы полезны тем, что помогают сэкономить время или выполнить какую-либо задачу без лишних усилий. Советы в книге — это приемы экономии времени или указатели на ресурсы, которые стоит попробовать, чтобы получить максимальную пользу от Python или при выполнении задач, связанных с алгоритмами и анализом данных.
 ВНИМАНИЕ	Мы не хотим выглядеть как рассерженные родители или какие-то маньяки, но вам следует избегать всего, что отмечено значком «Осторожно». В противном случае ваше приложение может работать не так, как ожидается, вы можете получить неверные ответы, казалось бы, от безотказных алгоритмов, или (в худшем случае) потерять данные.

 ПРИМЕЧАНИЕ	Когда вы видите этот значок, думайте о продвинутом совете или приеме. Возможно, эти крупитцы полезной информации покажутся вам невыносимо скучными, а может быть, они содержат именно то решение, которое нужно для запуска программы. Пропускайте эти фрагменты информации, когда захотите.
 ЗАПОМНИТЕ	Если не усвоите ничего другого из определенной главы или параграфа, то запомните материал, отмеченный этой иконкой. Такой текст обычно содержит описание важного процесса или информацию, которую нужно знать для успешной работы с Python или выполнения задач, связанных с алгоритмами и анализом данных.

Помимо книги

Эта книга — не конец вашего изучения Python и алгоритмов, а только начало. Мы предоставляем онлайн-контент, чтобы сделать книгу более гибкой и лучше отвечающей вашим потребностям. Таким образом, получая от вас электронные письма, мы можем отвечать на вопросы и рассказывать, как обновления Python или связанных с ним дополнений влияют на содержание книги. Фактически вы получаете доступ ко всем следующим полезным дополнениям:

- » **Шпаргалка.** Помните, как в школе вы использовали шпаргалки, чтобы получить лучшую оценку на тесте? Что ж, эта шпаргалка примерно такая же. Она предоставляет вам особые заметки о задачах, которые можно выполнять с помощью Python, Google Colab и алгоритмов, о которых знает не каждый. Чтобы найти шпаргалку к этой книге, перейдите на www.dummies.com и введите в поле поиска: «Algorithms For Dummies, 2nd Edition Cheat Sheet». Шпаргалка содержит действительно полезную информацию, например о том, как найти алгоритмы, которые обычно нужны для выполнения конкретных задач.
- » **Обновления.** Иногда происходят изменения. Например, мы могли не предвидеть грядущих изменений, когда заглядывали в свой хрустальный шар во время написания книги. Раньше это означало, что книга просто устаревала и становилась менее полезной; но теперь вы можете найти обновления к книге (если мы их сделаем), перейдя

на www.dummies.com и введя в поле поиска: «Algorithms For Dummies, 2nd Edition». Помимо этих обновлений, ознакомьтесь с публикациями в блоге, содержащими ответы на вопросы читателей и демонстрацию полезных приемов, связанных с книгой, по адресу <http://blog.johnmuellerbooks.com>.

» **Сопроводительные файлы.** Эй! Кто действительно хочет вручную набирать весь код из книги и воссоздавать все эти графики? Большинство читателей предпочитают тратить время на реальную работу с Python, выполнение задач с использованием алгоритмов и изучение интересных вещей, которые они могут делать, а не на набор текста. К счастью, примеры из книги доступны для скачивания, поэтому вам нужно просто прочитать книгу, чтобы изучить методы использования алгоритмов. Вы можете найти эти файлы, введя в поиск «Algorithms For Dummies, 2nd Edition» на сайте www.dummies.com и прокрутив левую часть открывшейся страницы. Исходный код также доступен по адресам: <http://www.johnmuellerbooks.com/source-code> и https://addons.eksmo.ru/it/algo4d_2ed-master.zip.

Куда двигаться дальше

Пора начать ваше путешествие в мир алгоритмов! Если вы не знакомы с алгоритмами, начните с главы 1 и продвигайтесь по книге в темпе, который позволит вам усвоить максимум материала. Обязательно ознакомьтесь с Python, так как в книге этот язык используется для примеров.

Если вы новичок и вам нужно как можно быстрее приступить к изучению алгоритмов, можете перейти сразу к главе 3, но имейте в виду, что некоторые темы позже могут показаться вам немного запутанными.

Читатели, которые уже знакомы с Python и установили соответствующие версии языка, могут сэкономить время и перейти сразу к главе 5. При возникновении вопросов всегда можно вернуться к предыдущим главам. Однако важно понимать принцип работы каждого метода, прежде чем переходить к следующему. Каждый прием, пример кода и процедура содержат важные уроки, и вы можете упустить жизненно важный материал, если начнете пропускать слишком много информации.

1

**НАЧАЛО
РАБОТЫ
С АЛГОРИТМАМИ**

В ЭТОМ РАЗДЕЛЕ

Определение алгоритмов и их разработка

Использование Google Colab для работы с алгоритмами

Выполнение основных операций с данными

Создание класса для работы с матрицами

В ЭТОЙ ГЛАВЕ

- » Определение того, что подразумевается под алгоритмом
- » Использование компьютеров для применения алгоритмов с целью получения решений
- » Определение того, чем задачи отличаются от решений
- » Выполнение операций с данными для поиска решения

Глава 1

Знакомство с алгоритмами

Если вы относитесь к большинству людей, то, вероятно, испытываете замешательство, открывая эту книгу и начиная свое приключение с алгоритмами, потому что многие тексты никогда не объясняют, что такое алгоритм, не говоря уже о том, зачем его использовать. Слушать об алгоритмах все равно что снова оказаться в школе с монотонно вещающим учителем: вы засыпаете от скуки, потому что алгоритмы сейчас не кажутся особенно полезными.

Первый параграф этой главы посвящен тому, чтобы помочь вам точно понять, что означает термин *алгоритм* и какую пользу вы получите от умения использовать алгоритмы. Алгоритмы вовсе не нечто таинственное: на самом деле, они используются повсеместно и вы, вероятно, годами пользовались ими или получали от них помощь, даже не подозревая об этом. Так что это скрытые знания! По правде говоря, алгоритмы становятся основой, которая поддерживает и регулирует все важное в нашем все более сложном и технологичном обществе.

Во втором параграфе главы рассматривается, как использовать компьютеры для создания решений задач с помощью алгоритмов, как различать проблемы и решения и что нужно делать для преобразования данных, чтобы найти решение. Цель состоит в том, чтобы помочь вам отличить алгоритмы от других задач, которые люди путают с алгоритмами. Короче говоря, вы узнаете, почему действительно важно разбираться в алгоритмах и как применять их к данным.

В третьем параграфе главы алгоритмы рассматриваются с практической точки зрения — через призму терминологии, используемой для их понимания и представления, что показывает: реальный мир часто далек от совершенства. Понимание того, как описывать алгоритм реалистичным образом, также помогает соотнести ожидания с тем, на что алгоритм действительно способен.

В заключительном параграфе главы обсуждаются данные. Алгоритмы, с которыми вы будете работать в книге, требуют входных данных в определенной форме, что иногда означает необходимость изменения данных в соответствии с требованиями алгоритма. Преобразование данных не меняет их содержания. Вместо этого оно меняет представление и форму данных так, чтобы алгоритм мог помочь вам увидеть новые закономерности, которые не были очевидны раньше (но фактически присутствовали в данных все время).

§ 1. Описание алгоритмов

Хотя люди решали алгоритмические задачи вручную на протяжении тысячелетий, это может требовать огромных затрат времени и множества числовых вычислений в зависимости от сложности решаемой задачи. Алгоритмы нужны для поиска решений, и чем они быстрее и проще, тем лучше. Существует огромный разрыв между математическими алгоритмами, исторически созданными гениями своего времени, такими как Евклид (<https://www.britannica.com/biography/Euclid-Greek-mathematician>), сэр Исаак Ньютон (<https://www.britannica.com/biography/Isaac-Newton>) или Карл Фридрих Гаусс (<https://www.britannica.com/biography/Carl-Friedrich-Gauss>), и современными алгоритмами, разработанными в университетах и частных исследовательских лабораториях. Главная причина такого разрыва — использование компьютеров. Применение компьютеров для решения задач с помощью подходящего алгоритма значительно ускоряет процесс. Вы можете заметить, что сегодня решения задач появляются быстрее — отчасти потому, что вычислительная мощность стала доступной и постоянно растет.

При работе с алгоритмами вы рассматриваете входные данные, желаемые результаты и процесс (последовательность действий), используемый для получения нужного результата из заданных входных данных. Однако можно неправильно понять терминологию и неверно представлять работу алгоритмов, если не учитывать, как они функционируют в реальных условиях.

Источники информации об алгоритмах часто представляют их запутанным образом, будучи слишком сложными или даже откровенно неверными. Хотя вы можете встретить и другие определения, в этой книге используются следующие определения терминов, которые люди часто путают с алгоритмами (хотя ими не являются):



ЗАПОМНИТЕ

- » **Уравнение** – числа и символы, которые в совокупности приравняются к определенному значению. Уравнение всегда содержит знак равенства, чтобы было понятно, что числа и символы представляют конкретное значение с другой стороны знака равенства. Уравнения обычно содержат переменную информацию, представленную в виде символов, хотя использование переменных необязательно.
- » **Формула** – комбинация чисел и символов, используемая для выражения информации или идей. Формулы обычно представляют математические или логические концепции, например – определение наибольшего общего делителя (НОД) двух целых чисел (видео по ссылке <https://www.khanacademy.org/math/cc-sixth-grade-math/cc-6th-factors-and-multiples/cc-6th-gcf/v/greatest-common-divisor> показывает, как это работает). Как правило, они показывают взаимосвязь двух или более переменных.
- » **Алгоритм** – последовательность шагов, используемая для решения задачи. Эта последовательность представляет собой уникальный метод решения проблемы, предлагая конкретное решение. Алгоритм необязательно должен представлять математические или логические концепции, хотя примеры в книге часто относятся именно к этим категориям, поскольку алгоритмы наиболее часто используются именно таким образом. Чтобы процесс можно было считать алгоритмом, он должен быть:
 - **конечным.** Алгоритм должен в итоге решить задачу. В книге рассматриваются задачи с известным решением, чтобы вы могли оценить, правильно ли алгоритм решает проблему;
 - **четко определенным.** Последовательность шагов должна быть точной и представлять собой понятные шаги. Особенно важно, – учитывая, что в использовании алгоритмов участвуют компьютеры, – чтобы компьютер мог понять эти шаги для создания работоспособного алгоритма;
 - **эффективным.** Алгоритм должен решать все случаи задачи, для которой он был определен. Алгоритм всегда должен решать поставленную перед ним задачу. Хотя следует предвидеть возможность некоторых сбоев (их вероятность крайне мала и возникает только в ситуациях, допустимых для предполагаемого использования алгоритма).

С учетом этих определений следующие блоки помогут прояснить точную природу алгоритмов. Цель состоит не в том, чтобы дать точное определение, а в том, чтобы помочь вам понять, как алгоритмы вписываются в общую картину, и сформировать собственное представление о том, что такое алгоритмы и почему они так важны.

Правильный способ приготовления тостов: определение применения алгоритмов

Алгоритм всегда представляет собой последовательность шагов и не обязательно выполняет эти шаги для решения математической формулы. Область применения алгоритмов невероятно широка. Существуют алгоритмы, решающие задачи в науке, медицине, финансах, промышленном производстве и снабжении, а также в сфере коммуникации. Алгоритмы поддерживают все аспекты повседневной жизни человека. Любую последовательность действий в нашей жизни, которая конечна, четко определена и эффективна, можно рассматривать как алгоритм. Например, даже такое тривиальное и простое действие, как приготовление тостов, можно превратить в алгоритм. Фактически процедура приготовления тостов часто рассматривается на занятиях по информатике, как описано на сайте <http://brianaspinall.com/now-thats-how-you-make-toast-using-computer-algorithms>.

К сожалению, алгоритм на этом сайте содержит недочеты. Инструктор никогда не вынимает хлеб из упаковки и не включает тостер в розетку, поэтому в результате получается испорченный обычный хлеб, все еще находящийся в упаковке и засунутый в неработающий тостер (подробнее см. в обсуждении на <http://blog.johnmuellerbooks.com/2013/03/04/procedures-in-technical-writing>). Тем не менее идея верна, хотя и требует некоторых небольших, но существенных корректировок, чтобы сделать алгоритм конечным и эффективным.

Одно из самых распространенных применений алгоритмов — это способ решения формул. Например, при работе с НОД двух целых чисел можно выполнить задачу вручную, перечислив все делители обоих чисел и выбрав наибольший общий делитель. Например, НОД (20, 25) равен 5, потому что 5 — это наибольшее число, которое делит без остатка и 20, и 25. Однако вычисление каждого НОД вручную отнимает много времени и чревато ошибками, поэтому греческий математик Евклид создал более эффективный алгоритм для выполнения этой задачи. Метод Евклида можно увидеть на <https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/the-euclidean-algorithm>.

Однако у одной формулы, представляющей собой набор символов и чисел для выражения информации или идей, может быть несколько решений, каждое из которых является алгоритмом. В случае с НОД другой распространенный алгоритм — это алгоритм, созданный Дерриком Генри Лемером (<https://www.imsc.res.in/~kapil/crypto/notes/node11.html>). Поскольку любую формулу можно решить разными способами, люди тратят много времени на сравнение алгоритмов, чтобы определить, какой из них лучше работает в конкретной ситуации. (Сравнение алгоритмов Евклида и Лемера можно найти по ссылке <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.31.693&rep=rep1&type=pdf>.)

Поскольку наше общество и сопутствующие технологии быстро меняются, нам нужны алгоритмы, способные поддерживать такой темп. Такие научные достижения, как расшифровка генома человека, стали возможны в наше время благодаря тому, что ученые нашли алгоритмы, работающие достаточно быстро для выполнения этой задачи. Определение того, какой алгоритм лучше в конкретной ситуации или при среднестатистическом использовании, действительно серьезный вопрос, который активно обсуждается среди специалистов по информатике.

Что касается компьютерных наук, у одного и того же алгоритма может быть несколько представлений; зачем делать что-то одним способом, когда можно придумать множество методов просто ради интереса? Например, алгоритм Евклида можно представить как в рекурсивной, так и в итеративной форме, как объясняется на <http://cs.stackexchange.com/questions/1447/what-is-most-efficient-for-gcd>. Короче говоря, алгоритмы представляют собой метод решения формул, но было бы ошибкой утверждать, что для любой формулы существует только один приемлемый алгоритм или что существует только одно приемлемое представление алгоритма. У использования алгоритмов для решения различных задач долгая история, это не то, что появилось только сейчас.

Даже если ограничиться областью информатики, науки о данных, искусственного интеллекта и других технических областей, можно найти множество видов алгоритмов — слишком много для одной книги. Например, *«Искусство программирования»* Дональда Э. Кнута (издательство Addison-Wesley) охватывает 3168 страниц в четырех томах (<http://www.amazon.com/exec/obidos/ASIN/0321751043/datacservip0f-20>) и все равно не исчерпывает тему (автор планировал написать еще несколько томов). Тем не менее вот несколько интересных примеров использования алгоритмов.

- » **Поиск.** Поиск информации или проверка того, что найденная информация – именно та, которую вы ищете, является важнейшей задачей. Без этой возможности вы не смогли бы выполнять многие задачи онлайн, например найти веб-сайт в интернете, продающий идеальную кофеварку для вашего офиса. Эти алгоритмы постоянно меняются, как показывает недавнее изменение алгоритма Google (<https://www.youaretech.com/blog/2021/1/26/webpage-experience-a-major-google-algorithmupdate-in-2021>).
- » **Сортировка.** Определение порядка представления информации крайне важно, поскольку большинство людей сегодня страдают от информационной перегрузки, и упорядочивание данных – один из способов справиться с этим потоком. Представьте, что вы зашли на Amazon, обнаружили там более тысячи кофеварок, но не можете отсортировать их по цене или количеству положительных отзывов. Более того, многие сложные алгоритмы требуют, чтобы данные были правильно упорядочены для надежной работы, поэтому сортировка – важное условие решения более сложных задач.
- » **Преобразование.** Конвертация данных из одного вида в другой критически важна для понимания и эффективного использования информации. Например, вы можете прекрасно разбираться в имперской системе мер, но все ваши источники используют метрическую систему. Перевод между двумя системами помогает вам понять данные.
- » **Планирование.** Обеспечение справедливого использования ресурсов – еще один способ, которым алгоритмы существенно проявляют себя. Например, светофоры на перекрестках уже не просто устройства, отсчитывающие секунды между сменой сигналов. Современные системы учитывают многие факторы, такие как время суток, погодные условия и интенсивность движения.
- » **Анализ графов.** У поиска кратчайшего пути между двумя точками много применений. Например, при решении задачи маршрутизации ваш GPS не смог бы функционировать без этого конкретного алгоритма, поскольку не мог бы направлять вас по городским улицам, используя самый короткий маршрут из точки А в точку Б. Хотя даже в этом

случае ваш GPS может направить вас напрямик в озеро (<https://theweek.com/articles/464674/8-drivers-who-blindly-followed-gps-into-disaster>).

- » **Криптография.** Обеспечение безопасности данных – это непрерывная битва с хакерами, постоянно атакующими источники информации. Алгоритмы позволяют анализировать данные, преобразовывать их в другую форму и затем возвращать в исходный вид.
- » **Генерация псевдослучайных чисел.** Представьте игры, которые никогда не меняются. Вы начинаете с того же места, выполняете те же шаги, в том же порядке каждый раз, когда играете. Без возможности генерировать числа, которые кажутся случайными, многие компьютерные задачи становятся невыполнимыми.



ЗАПОМНИТЕ

Этот список представляет собой невероятно краткий обзор. Люди используют алгоритмы для множества различных задач и разными способами, постоянно создавая новые алгоритмы для решения как существующих проблем, так и новых. Самое важное, что следует учитывать при работе с алгоритмами, — это то, что для определенного входа вы должны ожидать конкретного выхода. Второстепенные вопросы включают количество ресурсов, необходимых алгоритму для выполнения задачи, и время, требуемое для ее завершения. В зависимости от типа задачи и используемого алгоритма вам также может потребоваться учитывать вопросы точности и согласованности результатов.

Алгоритмы повсюду

Предыдущий блок упоминает алгоритм приготовления тостов по определенной причине. Почему-то приготовление тостов, вероятно, самый популярный алгоритм из когда-либо созданных. Многие младшеклассники пишут свои версии алгоритма приготовления тостов задолго до того, как могут решать даже самые простые математические задачи. Нетрудно представить, сколько существует вариаций алгоритма приготовления тостов и каков точный результат каждого из них. Результаты, вероятно, различаются в зависимости от человека и уровня проявленной креативности. Существуют даже веб-сайты, посвященные рассказам детям об алгоритмах, например <https://www.idtech.com/blog/algorithms-for-kids>. Короче говоря, алгоритмы встречаются в большом разнообразии и часто в неожиданных местах.

Каждая задача, которую вы выполняете на компьютере, включает алгоритмы. Некоторые алгоритмы являются частью аппаратного обеспечения компьютера. Сам процесс загрузки компьютера включает использование алгоритма. Вы также найдете алгоритмы в операционных системах, приложениях и любом другом программном обеспечении. Даже пользователи полагаются на алгоритмы. Скрипты помогают направлять пользователей выполнять задачи определенным образом, но эти же шаги могут появляться в виде письменных инструкций или как часть организационной политики.

Повседневные рутинные действия часто превращаются в алгоритмы. Подумайте о том, как вы проводите день. Если вы, как большинство людей, то выполняете, по сути, одни и те же задачи в одном и том же порядке каждый день, превращая его в алгоритм, который решает проблему того, как успешно жить, затрачивая при этом минимум энергии. В конце концов, рутина и делает именно это — превращает нас в эффективных людей.

На протяжении всей книги вы увидите одни и те же три элемента для каждого алгоритма:

- 1 описание проблемы;
- 2 создание последовательности шагов для решения задачи (четко определенных);
- 3 выполнение этих шагов для получения желаемого результата (конечного и эффективного).

§ 2. Использование компьютеров для решения проблем

Слово *компьютер* звучит довольно технически и, возможно, немного пугающе для некоторых людей, но сегодня мы погружены в компьютеры по уши (а может, и глубже). Вы постоянно носите с собой как минимум один компьютер — ваш смартфон. Если у вас есть какое-либо специальное устройство, например кардиостимулятор, оно тоже содержит компьютер. В современном автомобиле может быть до 150 компьютеров в виде встроенных микропроцессоров, которые регулируют расход топлива, работу двигателя, трансмиссию, рулевое управление и устойчивость (подробнее см. на <https://spectrum.ieee.org/software-eating-car>), обеспечивают работу усовершенствованных систем помощи водителю (ADAS) и содержат больше строк кода, чем истребитель. Компьютер существует для

того, чтобы решать задачи быстрее и с меньшими усилиями, чем при ручном решении. Поэтому неудивительно, что в книге мы используем еще больше компьютеров, чтобы помочь вам лучше понять алгоритмы.

Компьютеры различаются по многим параметрам. Компьютер в часах довольно маленький, а настольный — довольно большой. Суперкомпьютеры огромны и состоят из множества более мелких компьютеров, работающих вместе для решения сложных задач, таких как прогнозирование завтрашней погоды. Самые сложные алгоритмы опираются на специальные компьютерные возможности для решения задач, которые ставят перед ними люди. Да, можно использовать и меньшие ресурсы для выполнения задачи, но придется либо гораздо дольше ждать ответа, либо получать ответ, недостаточно точный для практического применения. В некоторых случаях ожидание затягивается настолько, что ответ уже теряет актуальность. С учетом необходимости обеспечения как скорости, так и точности, в следующих блоках рассматриваются некоторые специальные компьютерные функции, которые могут влиять на алгоритмы.

Максимальное использование возможностей современных ЦП и графических процессоров

Процессоры общего назначения (ЦП) изначально создавались как средство решения задач с помощью алгоритмов. Однако их универсальность также означает, что ЦП могут выполнять многие другие задачи, такие как перемещение данных или взаимодействие с внешними устройствами. Процессор общего назначения хорошо справляется со многими задачами, а значит, может выполнять шаги, необходимые для реализации алгоритма, но не обязательно быстро. Владельцы ранних ЦП могли добавлять в свои системы математические сопроцессоры (специальные математические чипы) для повышения производительности (подробнее см. на <https://www.computerhope.com/jargon/m/mathcopr.htm>). Сегодня математический сопроцессор встроен в ЦП, поэтому, приобретая процессор Intel i9, вы фактически получаете несколько процессоров в одном корпусе.

Графический процессор (ГПУ) — это специализированный процессор с возможностями, которые способствуют более быстрому выполнению алгоритмов. Многие думают, что ГПУ предназначены для того, чтобы принимать данные, особым образом их обрабатывать и затем отображать красивую картинку на экране. Однако любое компьютерное оборудование может служить более чем одной цели. Оказывается, ГПУ особенно хорошо подходят для выполнения преобразований данных, что во многих случаях является ключевой задачей при решении алгоритмических задач. Неудивительно, что люди, создающие алгоритмы, много времени

уделяют нестандартному мышлению. Это означает, что они часто находят способы решения проблем с помощью нетрадиционных подходов.

Суть в том, что центральные и графические процессоры служат наиболее часто используемыми чипами для выполнения алгоритмических задач. Первые хорошо справляются с задачами общего назначения, а вторые специализируются на поддержке математически интенсивных задач, особенно тех, которые связаны с преобразованием данных. Использование нескольких ядер делает возможной параллельную обработку (выполнение более одного алгоритмического шага одновременно). Добавление нескольких чипов увеличивает количество доступных ядер. Большее количество ядер повышает скорость, но некоторые факторы ограничивают прирост производительности. Использование двух процессоров i9 не даст двукратного увеличения скорости в сравнении с одним i9.

Работа со специализированными чипами

Математический сопроцессор и ГПУ — это два примера распространенных специализированных чипов, которые не используются для выполнения таких задач, как загрузка системы. Однако алгоритмы часто требуют использования необычных специализированных чипов для решения задач. Хотя это не книга об аппаратном обеспечении, стоит уделить время рассмотрению различных интересных чипов, таких как искусственные нейроны, над которыми работает Калифорнийский университет в Сан-Диего (UCSD) (<https://www.marktechpost.com/2021/06/03/ucsd-researchers-develop-an-artificial-neuron-device-that-could-reduce-energy-use-and-size-of-neural-network-hardware>). Представьте себе выполнение алгоритмической обработки с использованием устройств, имитирующих человеческий мозг. Это создало бы интересную среду для выполнения задач, которые сегодня могли бы быть невыполнимыми.

Нейронные сети — технология, используемая для имитации человеческого мышления и обеспечения возможности применения методов глубокого обучения в задачах машинного обучения, — сейчас получают преимущества от использования специализированных чипов (статья по адресу <https://analyticsindiamag.com/top-10-gpus-for-deep-learning-in-2021> описывает десять из них). Такие чипы не только выполняют алгоритмическую обработку с невероятной скоростью, но и обучаются в процессе выполнения задач, становясь еще быстрее с каждой итерацией. Обучающиеся компьютеры в итоге будут управлять роботами, способными самостоятельно (в определенной степени) передвигаться подобно роботам из фильма «Я, робот». Некоторые роботы уже умеют даже демонстрировать мимические выражения (<https://www.sciencedaily.com/releases/2021/05/210527145244.htm>).



ЗАПОМНИТЕ

Независимо от принципа работы, специализированные процессоры в итоге будут обеспечивать работу всевозможных алгоритмов, имеющих реальные последствия в мире. Многие из этих практических применений уже можно найти в относительно простой форме. Например, представьте задачи, которые должен решать робот для приготовления пищи, — переменные, которые он должен учитывать в режиме реального времени. Такой робот уже существует (это лишь один из множества примеров промышленных роботов, использующих алгоритмы для производства материальных товаров), и можно не сомневаться, что он опирается на алгоритмы, описывающие последовательность действий, а также на специальные чипы, обеспечивающие быстрое выполнение задач (<https://www.therobotreport.com/picnic-pizza-making-robot-is-now-available>).



ПРИМЕЧАНИЕ

В итоге может стать возможным использование человеческого разума в качестве процессора с выводом информации через специальный интерфейс. Когда-то люди экспериментировали с непосредственной имплантацией процессоров в мозг, но последняя инновация основана на использовании кровеносных сосудов для создания соединения (<https://www.independent.co.uk/life-style/gadgets-and-tech/brain-computer-interface-vein-als-stent-neuralink-b1556167.html>). Представьте систему, в которой люди могут решать задачи с помощью алгоритмов на скорости компьютеров, но с творческим потенциалом «что, если», присущим человеку.

Сети: совместное использование — это больше чем забота

Если у вас нет неограниченных финансовых средств, эффективное использование некоторых алгоритмов может оказаться невозможным даже при наличии специализированных чипов. В этом случае можно объединить компьютеры в сеть. Используя специальное программное обеспечение, один компьютер (хост) может задействовать процессоры всех клиентских компьютеров, на которых запущен *агент* (своего рода фоновое приложение, работающее в памяти и предоставляющее доступ к процессору). Используя такой подход, можно решать невероятно сложные задачи, распределяя части проблемы между несколькими клиентскими компьютерами. Когда каждый компьютер в сети решает свою часть задачи, он отправляет результаты обратно хосту, который собирает все фрагменты воедино для получения общего ответа. Такой метод называется *кластерными вычислениями*.

И это вовсе не научная фантастика — люди используют методы кластерных вычислений (<https://www.geeksforgeeks.org/an-overview-of-cluster-computing>) самыми разными интересными способами. Например, в статье <https://turingpi.com/12-amazing-raspberry-pi-cluster-use-cases> подробно описывается, как создать собственный суперкомпьютер (среди прочих задач), объединив несколько плат Raspberry Pi (<https://www.raspberrypi.org>) в единый кластер.

Распределенные вычисления — еще одна разновидность кластерных вычислений (но с более свободной организацией) — тоже популярны. Список проектов распределенных вычислений можно найти по адресу https://en.wikipedia.org/wiki/List_of_distributed_computing_projects. Список включает несколько крупных проектов, таких как программа поиска внеземного разума (SETI). Вы также можете пожертвовать свободные вычислительные мощности своего компьютера для работы над поиском лекарства от рака. Список потенциальных проектов впечатляет. Многие из них размещены на платформе Открытой инфраструктуры Беркли для сетевых вычислений (<https://boinc.berkeley.edu>), но есть и другие спонсоры.

Использование доступных данных

У части решения задач с помощью алгоритмов нет ничего общего с вычислительной мощностью, нестандартным мышлением или чем-либо физическим. Для создания решения большинства задач вам еще нужны данные, на основе которых можно сделать выводы. Например, в алгоритме приготовления тоста вам нужно знать о наличии хлеба, тостера, электричества для его работы и так далее, прежде чем вы сможете решить задачу приготовления тоста. Данные становятся важными, потому что вы не сможете завершить алгоритм, если отсутствует хотя бы один элемент необходимого решения. Конечно, вам могут понадобиться и дополнительные входные данные. Например, человек, желающий тост, может не любить ржаной хлеб. В этом случае, даже если у вас есть только ржаной хлеб, его наличие все равно не приведет к успешному результату.

Данные поступают из самых разных источников и в самых разных формах. Вы можете получать потоковые данные от источников вроде мониторов реального времени, обращаться к публичным источникам данных, использовать частные данные из баз данных, извлекать их с веб-сайтов или получать многими другими способами, которые слишком многочисленны, чтобы их здесь перечислять. Данные могут быть статическими (неизменными) или динамическими (постоянно меняющимися). Вы можете обнаружить, что данные полные или в них нет элементов. Данные могут быть представлены не в том виде (например, когда вы получаете

значения в английской системе мер, а для решения задачи с весом требуются метрические единицы). Данные могут быть в табличном формате, когда вам нужна другая форма. Они могут храниться в неструктурированном виде (например, в нереляционной базе данных или просто в куче разных файлов), когда вам нужна структура реляционной базы данных. Короче говоря, чтобы решать задачи с помощью алгоритма, вам нужно знать много деталей о данных, которые он использует.



ЗАПОМНИТЕ

Поскольку данные поступают во множестве форм и работать с ними приходится разными способами, в книге много внимания уделяется данным. Начиная с главы 6 вы узнаете, как структуры данных вступают в игру. Перейдя к главе 7, вы начнете изучать, как искать в данных то, что вам нужно. Главы 12–14 помогут вам работать с большими данными. Впрочем, информацию о специфике работы с данными можно найти практически в каждой главе книги, потому что без данных алгоритм не может решить ни одной задачи.

§ 3. Различение проблем и решений

В книге рассматриваются две составляющие алгоритмического взгляда на реальный мир. С одной стороны, у вас есть **проблемы**, которые нужно решить. Проблема может описывать желаемый результат работы алгоритма или препятствие, которое нужно преодолеть для получения этого результата. С другой стороны, у вас есть **решения** — методы, или шаги, используемые для решения проблем. Решение может относиться как к одному шагу, так и ко многим шагам внутри алгоритма. Фактически выходные данные алгоритма, ответ на последнем шаге, — это тоже решение. Следующие блоки помогут вам понять некоторые важные аспекты проблем и решений.

Корректность и эффективность

Использование алгоритмов нацелено на получение приемлемого ответа. Это обусловлено тем, что при обработке нечётко определённых входных данных некоторые алгоритмы способны генерировать несколько допустимых ответов. В жизни достижение абсолютно точного решения зачастую невозможно. Хотя точность остаётся идеальной целью, на практике достаточным считается результат, отвечающий установленным критериям приемлемости.

Получение максимально точного ответа может занять слишком много времени. Если стремиться к полной точности и тратить на это слишком много ресурсов, информация может потерять ценность, а время будет потрачено впустую. Выбор между двумя алгоритмами, решающими одну и ту же задачу, может сводиться к компромиссу между скоростью и точностью. Быстрый алгоритм не всегда дает точный ответ, но его результат может оказаться достаточно полезным.

Неправильные ответы могут создавать проблемы. Быстрое получение множества неверных ответов так же плохо, как и медленное получение множества абсолютно точных решений. Одна из целей книги — помочь вам найти золотую середину между слишком быстрым и слишком медленным, между неточным и излишне точным. Хотя ваш учитель математики подчеркивал необходимость давать правильный ответ именно так, как это излагалось в учебнике, в реальном мире математика часто связана с оценкой вариантов и принятием компромиссных решений, которые влияют на вас самым неожиданным образом.

Понимание, что бесплатный сыр только в мышеловке

Возможно, вы слышали распространенный миф о том, что можно получить от компьютера решение любой задачи, не прилагая особых усилий к поиску решения. К сожалению, не существует абсолютного решения ни для одной проблемы, а получение оптимальных ответов часто требует значительных затрат. Работая с алгоритмами, вы быстро обнаруживаете необходимость выделять дополнительные ресурсы, для получения быстрых и точных ответов. Размер и сложность используемых источников данных также сильно влияют на качество решения. По мере увеличения размера и сложности возрастает и потребность в дополнительных ресурсах.

Адаптация стратегии к проблеме

В разделе 5 рассматриваются стратегии, которые можно использовать для снижения затрат при работе с алгоритмами. Лучшие математики используют приемы, позволяющие получить больше результатов при меньших вычислительных затратах. Например, можно создать один сложный алгоритм для решения задачи или использовать набор более простых алгоритмов для решения той же задачи, но с применением нескольких процессоров. Набор простых алгоритмов обычно работает быстрее и лучше, чем один сложный алгоритм, хотя такой подход может показаться нелогичным.

Описание алгоритмов на универсальном языке

Алгоритмы действительно обеспечивают основу для общения между людьми, даже когда у них разные взгляды и они говорят на разных языках. Например, теорема Байеса (вероятность наступления события при определенных предпосылках; краткое объяснение этой удивительной теоремы можно найти по адресу <https://betterexplained.com/articles/an-intuitive-and-short-explanation-of-bayes-theorem>):

$$P(B|E) = P(E|B) \cdot P(B) / P(E)$$

выглядит одинаково, независимо от того, говорите ли вы на английском, испанском, китайском, немецком, французском или любом другом языке. Какой бы язык вы ни использовали, алгоритм будет выглядеть и работать одинаково при одних и тех же входных данных. Алгоритмы помогают преодолеть всевозможные барьеры, разделяющие людей, выражая идеи в форме, которую каждый может проверить. Читая эту книгу, вы откроете для себя красоту и магию алгоритмов в передаче даже самых тонких мыслей другим людям.



СОВЕТ

Помимо универсальных математических обозначений, алгоритмы используют языки программирования как средство объяснения и передачи решаемых ими формул. Алгоритмы можно найти на C, C++, Java, Fortran, Python (как в этой книге) и других языках. *Псевдокод* — это способ описания компьютерных операций с помощью обычных слов. Некоторые авторы используют псевдокод, чтобы обойти тот факт, что алгоритм может быть предоставлен на языке программирования, которого вы не знаете. Кроме того, псевдокод зачастую более лаконичный, чем язык программирования, поскольку позволяет использовать интуитивно понятные идеи, которые язык программирования может не передать должным образом.

Решение проблем, похожих на кирпичные стены, только сложнее

Важно понимать, что алгоритмы можно использовать для решения задач любой сложности. Алгоритм не думает, не испытывает эмоций и не заботится о том, как вы его используете (или даже злоупотребляете им). Алгоритмы можно применять любым способом, необходимым для решения проблемы. Например, одна и та же группа алгоритмов, применяемая для распознавания лиц в качестве альтернативы компьютерным паролям (в целях безопасности), может использоваться для обнаружения террористов, скрывающихся

в аэропорту или для распознавания потерявшегося ребенка, бродящего по улицам. Одни и те же алгоритмы могут применяться по-разному; как их использовать, зависит от интересов пользователя. Одна из целей книги — продемонстрировать, что для решения сложных проблем иногда достаточно простого алгоритма.

§ 4. Структурирование данных для получения решения

Люди мыслят о данных неспецифическим образом и применяют различные правила к одним и тем же данным, чтобы понять их так, как компьютеры никогда не смогут. Взгляд компьютера на данные структурирован, прост, бескомпромиссен и определенно некреативен. Когда люди готовят данные для использования компьютером, данные часто взаимодействуют с алгоритмами неожиданным образом и производят нежелательный результат. Проблема заключается в том, что человек не понимает ограниченное представление данных. В следующих блоках описываются два аспекта данных, которые будут проиллюстрированы во многих последующих главах.

Понимание точки зрения компьютера

У компьютера простой взгляд на данные, но люди обычно не понимают его. Во-первых, для компьютера все является числом, поскольку компьютеры не предназначены для работы с какими-либо другими типами данных. Люди видят символы на экране компьютера и предполагают, что компьютер взаимодействует с данными таким же образом; но компьютер не понимает данные или их значение. Буква *A* для компьютера — это просто число 65. На самом деле, это даже не совсем число 65. Компьютер видит последовательность электрических импульсов, которая соответствует бинарному значению 01000001.

Компьютеры также не понимают концепцию прописных и строчных букв. Для человека строчная буква *a* — это просто другая форма прописной *A*, но для компьютера это два разных значения. Строчная буква *a* представлена числом 97 (бинарное значение 01100001).

Если такие простые сравнения отдельных букв могут вызывать подобные проблемы между людьми и компьютерами, нетрудно представить, что происходит, когда люди начинают делать слишком много предположений о других типах данных. Например, компьютер не может слышать или ценить музыку. Тем не менее музыка звучит из компьютерных колонок. То же самое верно и для графики. Компьютер видит последовательность нулей и единиц, а не изображение с красивым сельским пейзажем.



ЗАПОМНИТЕ

При работе с алгоритмами важно рассматривать данные с точки зрения компьютера. Компьютер видит только нули и единицы, ничего больше. Следовательно, когда вы начинаете прорабатывать требования алгоритма, вы должны смотреть на данные именно таким образом. Вы можете даже обнаружить, что такой компьютерный взгляд на данные делает поиск некоторых решений проще, а не сложнее. По мере чтения книги вы узнаете больше об этой особенности восприятия данных.

Упорядочивание данных имеет значение

У компьютеров также есть строгое представление о форме и структуре данных. Когда вы начинаете работать с алгоритмами, то обнаруживаете, что значительная часть работы заключается в представлении данных в форме, которую компьютер может использовать при применении алгоритма для поиска решения проблемы. Хотя человек способен мысленно распознавать закономерности в данных, которые не совсем точно упорядочены, компьютерам действительно нужна точность, чтобы найти ту же закономерность. Преимущество такой точности в том, что компьютеры часто могут делать видимыми новые закономерности. Фактически это одна из главных причин использования алгоритмов с компьютерами — помочь обнаружить новые закономерности, а затем использовать их для выполнения других задач. Например, компьютер может распознать модель покупательского поведения клиента, чтобы вы могли использовать эту информацию для автоматического увеличения продаж.

В ЭТОЙ ГЛАВЕ

- » Рассмотрение способов решения задачи
- » Использование метода «разделяй и властвуй» при решении задач
- » Понимание жадного подхода к решению задач
- » Определение стоимости решений задачи
- » Проведение оценки алгоритмов

ГЛАВА 2

Рассматриваем проектирование алгоритмов

Алгоритм состоит из последовательности шагов, используемых для решения задачи, которая может включать входные данные, служащие основой для решения проблемы, и иногда — ограничения, которые любое решение должно учитывать, прежде чем алгоритм будет считаться эффективным. Первый параграф этой главы поможет вам понять необходимость создания алгоритмов, которые являются одновременно гибкими (могут обрабатывать широкий спектр входных данных) и эффективными (дают желаемый результат).

Во втором параграфе главы рассматривается, как найти решение задачи. Чувство растерянности перед сложной проблемой — обычное явление, и самый распространенный способ справиться с ним — разделить проблему на более мелкие, управляемые части. Такой подход «разделяй и властвуй» изначально применялся в военном деле (историю подхода можно найти по ссылке <https://classroom.synonym.com/civilization-invented-divide-conquer-strategy-12746.html>). Однако те же идеи люди используют для упрощения проблем любого рода.



ЗАПОМНИТЕ

В третьем параграфе главы речь идет о жадном подходе к решению задач. Обычно у жадности негативный оттенок (например, когда вы крадете фруктовый десерт с тарелки друга), но не в этом случае. *Жадный алгоритм* — это алгоритм, который делает оптимальный выбор на каждом этапе решения задачи. Таким образом он стремится получить общее оптимальное решение проблемы. К сожалению, эта стратегия не всегда работает, но попробовать ее все же стоит. Нередко она приводит к достаточно полезному результату.

Независимо от выбранного подхода к решению, каждый алгоритм связан с определенными затратами. Будучи рачительными покупателями, люди, активно использующие алгоритмы, хотят получить наилучшую сделку, что подразумевает проведение анализа затрат и выгод. Конечно, получение лучшей сделки также предполагает, что человек, использующий алгоритм, представляет, какое решение является достаточно хорошим. Получение слишком точного решения (предлагающего избыточные детали) или решения, дающего слишком много выходных данных, часто оказывается расточительным, поэтому часть контроля над затратами заключается в получении необходимых результатов и больше ничего.

Чтобы понять, что представляет собой алгоритм, нужно уметь измерять его различными способами. Измерения создают в вашем сознании картину удобства использования, размера, потребления ресурсов и стоимости. Что еще важнее, измерения дают возможность проводить сравнения. Без измерений невозможно сравнивать алгоритмы. А если вы не можете сравнить алгоритмы, то не сможете выбрать лучший для конкретной задачи.

§ 1. Начинаем решать проблему

Прежде чем решать любую проблему, вы должны ее понять. И дело не только в оценке масштаба проблемы. Знание того, что у вас есть определенные входные данные и требуются определенные выходные данные, — это начало, но этого недостаточно для создания решения. Часть процесса поиска решения заключается в том, чтобы:

- » узнать, как другие люди находили новые решения проблем;
- » понимать, какие ресурсы имеются;
- » определить, какие решения работали для подобных проблем в прошлом;
- » рассмотреть, какие решения не давали желаемого результата.

Имейте в виду, что необязательно выполнять эти этапы по порядку, а иногда и вовсе придется возвращаться к какому-либо этапу после получения дополнительной информации. Процесс начала решения проблемы итеративный: вы продолжаете работать, пока не достигнете понимания сути проблемы.

Моделирование реальных задач

Реальные задачи отличаются от тех, что встречаются в учебниках, в основном потому, что реальный мир начисто лишен воображения. Создавая учебник, автор часто придумывает простой пример, чтобы помочь читателю понять базовые принципы работы. Такой пример моделирует лишь один аспект более сложной проблемы. Реальная же задача может потребовать комбинирования нескольких методов для создания полного решения. Например, чтобы найти наилучший ответ, вам может потребоваться:

- 1 отсортировать набор ответов по определенному критерию,
- 2 выполнить некоторую фильтрацию и преобразование,
- 3 произвести поиск в полученном результате.

Без этой последовательности шагов адекватное сравнение каждого из ответов может оказаться невозможным, и вы получите неоптимальный результат. Набор алгоритмов, используемых совместно для достижения желаемого результата, называется *ансамблем*. Об их применении в машинном обучении можно прочесть в книге «*Машинное обучение для чайников*» (2-е издание, авторы Джон Пол Мюллер и Лука Массарон, издательство Wiley). Статья по адресу <https://machinelearningmastery.com/tour-of-ensemble-learning-algorithms> дает краткий обзор принципов работы ансамблей.

Однако реальные задачи сложнее, чем просто работа со статическими данными или их однократная обработка. Например, любой движущийся объект, будь то автомобиль, самолет или робот, постоянно получает входные данные. Каждое обновление входных данных включает информацию об ошибках, которую реальное решение должно учитывать в результате, чтобы эти машины продолжали работать правильно. Помимо прочих алгоритмов, для постоянных вычислений требуется пропорционально-интегрально-дифференциальный (ПИД) алгоритм (подробное объяснение этого алгоритма см. на <https://www.ni.com/en-us/innovations/white-papers/06/pid-theory-explained.html>), который управляет машиной с помощью цепи обратной связи. Каждое вычисление делает реше-

ние, используемое для управления машиной, точнее, именно поэтому машины часто проходят стадию стабилизации при первом включении.

При моделировании реальной задачи нужно учитывать, что могут возникнуть и неочевидные проблемы. Решение, даже основанное на серьезных математических выкладках и надежной теории, может не сработать. Например, во время Второй мировой войны у союзников была серьезная проблема с потерями бомбардировщиков. Инженеры проанализировали каждое пулевое отверстие в каждом вернувшемся самолете. После анализа они использовали полученное решение для усиления бронирования союзных самолетов, чтобы обеспечить возвращение большего их числа. Это не сработало. И тут появился Абрахам Вальд. Этот математик предложил неочевидное решение: установить броню во всех местах, где не было пулевых отверстий (поскольку зоны с пулевыми отверстиями уже достаточно прочны, иначе самолет не вернулся бы). Полученное решение сработало и теперь используется как основа для понятия *систематической ошибки выжившего* (факт того, что выжившие в инциденте часто не показывают, что на самом деле привело к потерям) при работе с алгоритмами. Подробнее об этом увлекательном историческом факте можно прочитать на <https://medium.com/@penguinpress/an-excerpt-from-how-not-to-be-wrong-by-jordan-ellenberg-664e708cfc3d>.

Моделирование реальных задач может включать и добавление того, что ученые обычно считают нежелательными характеристиками. Например, ученые часто считают шум нежелательным, поскольку он скрывает исходные данные. Возьмем слуховой аппарат, который удаляет шум, чтобы человек мог лучше слышать (подробности можно найти по ссылке <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4111515>). Есть много методов устранения шума, и некоторые из них вы найдете в этой книге начиная с главы 9, в рамках обсуждения других тем.

Однако, как бы парадоксально это ни звучало, добавление шума в данные тоже требует алгоритма, который обеспечивает полезный результат при наличии этого шума. Например, Кен Перлин в 1983 году хотел избавиться от машинного вида компьютерной графики и создал для этого алгоритм. Результатом стал шум Перлина (подробности можно найти по ссылке <https://catlikecoding.com/unity/tutorials/pseudorandom-noise/perlin-noise>). Эффект оказался настолько полезным, что Перлин получил за работу премию Академии киноискусств (подробнее по ссылке <https://cs.nyu.edu/~perlin/doc/oscar.html>). Реальные сценарии часто требуют решений, которые могут быть неочевидны при работе в лаборатории или в процессе обучения.



ЗАПОМНИТЕ

Главная суть этого блока в том, что для создания решений часто требуется несколько итераций, возможно, придется потратить много времени на их доработку, а очевидные решения могут вообще не сработать. При моделировании реальной задачи вы действительно начинаете с решений, найденных в учебниках, но затем должны выйти за рамки теории, чтобы найти настоящее решение проблемы. По мере изучения этой книги вы познакомитесь с широким спектром алгоритмов — все они помогут вам находить решения. Важно помнить, что вам может потребоваться комбинировать эти примеры различными способами и находить методы взаимодействия с данными так, чтобы это способствовало выявлению закономерностей, соответствующих требуемому результату.

Поиск решений и контрпримеров

В предыдущем блоке мы познакомились с особенностями поиска реальных решений, учитывающих проблемы, которые невозможно предусмотреть в лабораторных условиях. Однако просто найти решение — даже хорошее — недостаточно, поскольку оно тоже иногда дает сбой. Важная часть начального этапа решения проблемы — искать контрпримеры, выступая в роли адвоката дьявола. Контрпримеры нужны для того, чтобы:

- » потенциально опровергнуть решение;
- » лучше определить границы применимости решения;
- » рассмотреть ситуации, в которых гипотезы, лежащие в основе решения, остаются непроверенными;
- » понять ограничения решения.

Распространенный сценарий, иллюстрирующий решение и контрпример, — утверждение, что все простые числа нечетные (*простые числа* — это положительные целые числа, которые делятся нацело только на себя и на 1). Конечно, число 2 — простое, но оно четное, что делает исходное утверждение ложным. Тот, кто сделал это утверждение, мог бы уточнить его, сказав, что все простые числа — нечетные, кроме 2. Частичное решение задачи поиска всех простых чисел заключается в том, что нужно искать нечетные числа, за исключением случая с 2, которое четное. Во втором случае опровергнуть решение уже невозможно, но дополнение к исходному утверждению устанавливает границу.

Подвергая сомнению исходное утверждение, можно рассмотреть и ситуации, в которых гипотеза «все простые числа, кроме 2, нечетные» может не выполняться. Например, 1 — нечетное число, но не считается простым (подробнее см. на <https://primes.utm.edu/notes/faq/one.html>). Таким образом, теперь у исходного утверждения есть две границы, и его нужно переформулировать следующим образом: простые числа — больше 1 и обычно нечетные, за исключением 2, которое четное. Границы для простых чисел лучше определяются путем поиска и рассмотрения контрпримеров.



СОВЕТ

По мере роста сложности задачи растет и вероятность нахождения контрпримеров. Важное правило, которое следует учитывать: как и в случае с надежностью, чем больше точек отказа, тем выше вероятность возникновения сбоя. Важно рассматривать алгоритмы именно с этой точки зрения. Ансамбли простых алгоритмов могут давать лучшие результаты с меньшим количеством потенциальных контрпримеров, чем один сложный алгоритм.

Стоя на плечах гигантов

Существует миф, не поддающийся объяснению, о том, что методы, используемые сегодня для обработки огромных объемов данных, — это якобы что-то новое. Да, новые алгоритмы появляются постоянно, но их основой служат все предшествующие алгоритмы. Даже когда мы думаем о сэре Исааке Ньютоне как о человеке, создавшем что-то новое, стоит вспомнить его собственные слова (приведенные с оригинальным написанием того времени): «Если я видел дальше других, то потому, что стоял на плечах Гигантов» (дополнительные цитаты и размышления можно найти на https://en.wikiquote.org/wiki/Isaac_Newton).

Алгоритмы, которые мы используем сегодня, не были новыми даже во времена Аристотеля (см. на <https://plato.stanford.edu/entries/aristotle-mathematics>) и Платона (см. на https://www.storyofmathematics.com/greek_plato.html). Истоки современных алгоритмов настолько глубоко скрыты в истории, что максимум, что можно сказать, — математика опирается на адаптированные знания древних времен. Использование алгоритмов с античных времен должно внушать чувство уверенности, ведь современные алгоритмы основаны на знаниях, проверенных тысячелетиями.

Это не значит, что некоторые математики не переворачивали все с ног на голову за эти годы. Например, теория Джона Нэша — равновесие Нэша — существенно изменила современный взгляд на экономику (см. на <https://www.masterclass.com/articles/nash-equilibrium-explained>).

Конечно, признание такой работы приходит медленно. Нэшу пришлось долго ждать, прежде чем его работа получила серьезное профессиональное признание (подробнее об этом можно прочитать на <https://www.princeton.edu/main/news/archive/S42/72/29C63/index.xml>), несмотря на то что он получил Нобелевскую премию по экономике за свой вклад. Если вам интересно, история Джона Нэша показана в фильме «Игры разума», который содержит несколько спорных сцен, включая ту, где утверждается, что равновесие Нэша каким-то образом опровергает некоторые работы Адама Смита — другого выдающегося экономиста (одно из таких обсуждений можно найти на <https://www.quora.com/Was-Adam-Smith-wrong-as-claimed-by-John-Nash-in-the-movie-A-Beautiful-Mind>).

§ 2. Разделяем и властвуем

Если бы решение проблем было простым делом, этим занимался бы каждый. Однако мир до сих пор полон нерешенных задач, и эта ситуация вряд ли изменится в ближайшее время по одной простой причине: проблемы часто кажутся настолько масштабными, что невозможно даже представить их решение. Древние воины сталкивались с похожей проблемой. Армия противника могла казаться настолько большой, а их собственные силы — настолько малыми, что задача победы в войне представлялась невообразимо сложной, возможно, даже невыполнимой. Тем не менее, разделив вражескую армию на мелкие части и атакуя ее по частям, небольшая армия потенциально могла победить гораздо более сильного противника. (Древние греки, римляне и Наполеон Бонапарт были великими мастерами стратегии «разделяй и властвуй»; подробности можно найти в книге *«Наполеон Бонапарт для чайников»* Дж. Дэвида Маркхэма [издательство Wiley].)

Вы сталкиваетесь с той же проблемой, что и древние воины. Часто имеющиеся в вашем распоряжении ресурсы кажутся весьма скромными и недостаточными. Однако, разделив огромную проблему на мелкие части, которые можно понять по отдельности, вы в итоге сможете создать решение, работающее для всей проблемы в целом. Этот принцип лежит в основе алгоритмов: использовать пошаговый подход для решения проблем по частям. В следующих блоках мы подробнее рассмотрим подход «разделяй и властвуй» к решению задач.

Избегаем решений методом полного перебора

Решение методом полного перебора — способ, при котором вы проверяете каждый возможный ответ поочередно, чтобы найти наилучший вариант. (Его еще называют *исчерпывающим подходом*, в основном потому, что к его завершению вы будете полностью измотаны.) Безусловно, это тщательный метод, но в большинстве случаев он попусту тратит время и ресурсы. Проверка каждого ответа — даже когда легко доказать, что у конкретного варианта нет шансов на успех, — расходует время, которое алгоритм мог бы потратить на ответы с большей вероятностью успеха. Кроме того, проверка различных вариантов таким способом обычно расточительно расходует ресурсы, например память. Представьте это так: вы хотите подобрать комбинацию для замка и начинаете с «0, 0, 0», хотя знаете, что у этой конкретной комбинации нет шансов на успех, учитывая физические характеристики кодовых замков. Решение методом полного перебора все равно продолжит проверять комбинацию «0, 0, 0», а затем перейдет к столь же бессмысленной «0, 0, 1».



ЗАПОМНИТЕ

У каждого типа решений есть свои преимущества, хотя иногда весьма незначительные. У решения методом полного перебора есть одно такое преимущество. Поскольку вы проверяете каждый ответ, вам не нужно выполнять какую-либо предварительную обработку. Однако время, сэкономленное на пропуске предварительной обработки, вряд ли когда-либо окупит время, потраченное на проверку каждого ответа. Тем не менее вы можете использовать решение методом полного перебора, когда:

- » критически важно найти идеальное решение, если оно существует;
- » размер задачи ограничен;
- » можно использовать эвристики для уменьшения множества решений;
- » простота реализации важнее скорости.

Принцип KISS (Keeping it simple, silly, или «не усложняй»)

У решения методом полного перебора, описанного в предыдущем блоке, есть серьезный недостаток. Оно рассматривает всю проблему целиком. Это похоже на поиск книги в библиотеке, когда вы просматриваете полки книгу за книгой, даже не рассматривая какой-либо способ упростить поиск. Подход «разделяй и властвуй» к поиску книг иной. В этом случае вы начинаете с разделения библиотеки на детский и взрослый разделы. После этого вы делите взрослый раздел на категории. Наконец, вы ищете только в той части категории, которая содержит интересующую вас книгу. Именно для этого существуют такие системы классификации, как Десятичная система Дьюи (см. на <https://mcpl.info/childrens/how-use-dewey-decimal-system>). Суть в том, что метод «разделяй и властвуй» упрощает задачу.

Часть «разделяй» в этом методе также является важным способом лучше понять проблему. Попытка осмыслить планировку всей библиотеки может оказаться сложной задачей. Однако, если вы знаете, что нужная вам книга по сравнительной психологии находится в Класе 100, Разделе 150, Секции 156, ваша задача становится проще. Вы можете решить эту более узкую проблему, поскольку знаете, что каждая книга в Секции 156 будет содержать что-то по интересующей вас теме. Алгоритмы работают так же. Упрощая проблему, вы можете создать набор более простых шагов для поиска решения, что сокращает время поиска, уменьшает количество используемых ресурсов и повышает шансы найти именно то решение, которое вам нужно.

Разделение проблемы обычно эффективнее

После того как вы разделили проблему на управляемые части, нужно справиться с конкретной частью. Это означает создание точного определения проблемы. Вам нужна не просто любая книга по сравнительной психологии, а именно та, что написана Джорджем Романесом. То, что нужная книга находится в Секции 156 по системе Дьюи, — хорошее начало, но это не решает проблему. Теперь вам нужен просмотр каждой книги в Секции 156 для поиска конкретной книги. Процесс может пойти еще дальше — до поиска книги с определенным содержанием. Чтобы сделать этот процесс жизнеспособным, вы должны полностью разбить проблему, точно определить, что вам нужно, а затем, после того как вы полностью поймете проблему, использовать правильный набор шагов (алгоритм) для поиска того, что вам нужно.

§ 3. Узнаем, что жадность может быть полезной

В некоторых случаях невозможно понять, где конец процесса решения, и побеждаете ли вы вообще в этой битве. Все, что вы действительно можете сделать, — это обеспечить победу в отдельных сражениях, стремясь шаг за шагом приблизиться к победе в войне. Жадный метод использует именно этот подход: он ищет общий успех, выбирая наилучший возможный результат на каждом этапе.



ЗАПОМНИТЕ

Кажется, что победа в каждом сражении обязательно означает победу в войне, но в реальном мире это не всегда так. *Пиррова победа* — это когда кто-то выигрывает каждое сражение, но в итоге проигрывает войну, потому что цена победы превышает выгоду от выигрыша. Вы можете прочитать о пяти пирровых победах на <https://www.history.com/news/5-famous-pyrrhic-victories>. Эти истории показывают, что жадный алгоритм часто работает, но не всегда, поэтому нужно рассматривать наилучшее общее решение проблемы, а не ослепляться промежуточными победами. В следующих блоках описывается, как избежать пирровой победы при работе с алгоритмами.

Применение жадного подхода

Жадный подход часто используется как часть процесса оптимизации. Алгоритм рассматривает задачу пошагово и фокусируется только на текущем шаге. Каждый жадный алгоритм основан на двух предположениях:

- » на каждом шаге можно сделать единственный оптимальный выбор;
- » выбирая оптимальное решение на каждом шаге, вы можете найти оптимальное решение для всей задачи в целом.

Существует много жадных алгоритмов, каждый из которых оптимизирован для выполнения определенных задач. Вот несколько распространенных примеров, используемых для анализа графов (подробнее о графах в главе 9) и сжатия данных (подробнее о сжатии данных в главе 14), а также причины, по которым их стоит использовать.

- » **Алгоритм Краскала для построения минимального остовного дерева (МОД).** Алгоритм выбирает ребро между двумя узлами с наименьшим весом, а не с наибольшим, как можно было бы предположить из слова *жадный*. Такой алгоритм может помочь найти кратчайший путь между двумя точками на карте или выполнить другие задачи, связанные с графами.
- » **Алгоритм Прима для МОД.** Этот алгоритм разделяет неориентированный граф (в котором направление не учитывается) пополам. Затем выбирает ребро, соединяющее две половины так, чтобы суммарный вес двух половин был минимальным. Этот алгоритм можно использовать, например, в игре-лабиринте для поиска кратчайшего расстояния между началом и концом лабиринта.
- » **Кодирование Хаффмана.** Алгоритм присваивает код каждому уникальному элементу данных в потоке, причем наиболее часто встречающийся элемент получает самый короткий код. Например, при сжатии английского текста буква Е обычно получает самый короткий код, поскольку она используется чаще любой другой буквы алфавита. Изменяя метод кодирования, можно сжать текст и значительно уменьшить его размер, сократив время передачи.

Поиск хорошего решения

Ученые и математики так часто используют жадные алгоритмы, что им посвящена вся глава 15. Однако на самом деле вам нужно *хорошее* решение, а не просто какое-то конкретное. В большинстве случаев *хорошее* решение обеспечивает оптимальные результаты, которые можно измерить, но у слова хорошее может быть много значений в зависимости от предметной области. Нужно определить, какую проблему вы хотите решить и какое решение наилучшим образом отвечает вашим потребностям. Например, в инженерном деле может потребоваться оценить решения с учетом веса, размера, стоимости или других параметров либо, возможно, некоторой комбинации всех этих показателей, удовлетворяющей определенным требованиям.

Чтобы проиллюстрировать эту проблему, представьте, что вы создаете монетный аппарат, который выдает сдачу определенными суммами, используя минимально возможное количество монет (например, как часть автоматической кассы в магазине). Причина использования

минимального количества монет — снижение износа оборудования, уменьшение веса необходимых монет и сокращение времени выдачи сдачи (ведь ваши покупатели всегда спешат). Жадный подход решает задачу используя монеты максимально возможного номинала. Например, чтобы выдать сдачу в 16 центов, используется один дайм (10 центов), один никель (5 центов) и один пенни (1 цент).



ЗАПОМНИТЕ

Проблема возникает, когда нельзя использовать все типы монет для получения решения. Например, в аппарате могут закончиться никели. Чтобы выдать сдачу в 40 центов, жадное решение начнет с квотера (25 центов) и дайма (10 центов). К сожалению, никелей нет, поэтому монетный аппарат затем выдает пять пенни (5×1 цент), что в сумме дает семь монет. Оптимальным решением в этом случае было бы использование четырех даймов (4×10 центов). Таким образом, жадный алгоритм предоставляет определенное решение, но неоптимальное. Проблема размена привлекает значительное внимание из-за сложности ее решения. Дополнительные обсуждения можно найти в работах «Комбинаторика задачи о размене монет» Анны Адамашек и Михала Адамашека (<https://www.sciencedirect.com/science/article/pii/S0195669809001292>) и «Размен монет» Маюха Синха (<https://www.geeksforgeeks.org/coin-change-dp-7>).

§ 4. Вычисление затрат и последующие эвристики

Даже когда вы находите хорошее решение, которое одновременно эффективно и результативно, вам все равно нужно точно знать, во что оно обходится. Может оказаться, что стоимость использования конкретного решения все еще слишком высока, даже с учетом всех остальных факторов. Возможно, ответ получается почти вовремя, но не совсем, или используется слишком много вычислительных ресурсов. Поиск хорошего решения включает создание среды, в которой вы можете полностью протестировать алгоритм, создаваемые им состояния, операторы, используемые для изменения этих состояний, и время, необходимое для получения решения.

Часто оказывается, что *эвристический подход*, основанный на самообучении и дающий достаточно полезные результаты (необязательно оптимальные, но приемлемые), — это именно тот метод, который вам действительно нужен для решения задачи. Когда алгоритм выполняет часть необходимой работы за вас, это экономит время и усилия, поскольку вы можете создавать алгоритмы, которые видят закономерности лучше, чем люди. Следовательно, самообучение — это процесс, позволяющий

алгоритму показать вам потенциально полезный путь к решению (но вы все равно должны полагаться на человеческую интуицию и понимание, чтобы знать, является ли решение правильным). В следующих блоках описываются методы, которые вы можете использовать для вычисления стоимости алгоритма, используя эвристику как способ определения реальной полезности любого конкретного решения.

Представление задачи в виде пространства

Пространство задачи — это среда, в которой происходит поиск решения. Пространство представлено набором состояний и операторов, изменяющих эти состояния. Рассмотрим, например, игру с восемью плитками в рамке 3×3 . На каждой плитке изображена часть картинки, и плитки вначале расположены в случайном порядке, так что картинка перемешана. Цель состоит в том, чтобы, перемещая по одной плитке за раз, расположить их в правильном порядке и собрать картинку. Пример такой головоломки можно увидеть на сайте <https://www.proprofsgames.com/puzzle/sliding>.

Комбинация начального состояния (перемешанных плиток) и целевого состояния (плитки в определенном порядке) представляет собой *конкретную задачу*. Головоломку можно представить графически с помощью *графа пространства задачи*. Каждый узел такого графа отображает определенное состояние (восемь плиток в конкретном положении). Ребра представляют операции, например перемещение плитки номер восемь вверх. Когда вы перемещаете восьмую плитку вверх, картинка меняется — происходит переход в другое состояние.

Выигрыш в игре путем перехода из начального состояния в целевое — не единственный критерий. Чтобы решить головоломку эффективно, нужно выполнить задачу за минимальное количество ходов, то есть использовать наименьшее число операций. Минимальное количество ходов, необходимое для решения головоломки, называется *глубиной задачи*.

При представлении задачи в виде пространства нужно учитывать несколько факторов. Например, нужно принять во внимание максимальное количество узлов, которые поместятся в памяти, и то, соответствует ли объем доступной памяти ожидаемому количеству узлов, необходимому для решения задачи, что определяет *пространственную сложность*. Когда невозможно разместить все узлы в памяти одновременно, компьютер должен генерировать их только при необходимости, удаляя предыдущие узлы для освобождения памяти или сохраняя некоторые узлы в других местах. Чтобы определить, поместятся ли узлы в памяти, нужно учитывать *временную сложность*, поскольку более дли-

тельное выполнение алгоритма определяет максимальное количество узлов, создаваемых для решения задачи. Кроме того, важно учитывать *коэффициент ветвления* — среднее число узлов, создаваемых на каждом шаге в графе пространства задачи. При одинаковом решении алгоритм с более высоким коэффициентом ветвления создаст больше узлов, чем алгоритм с меньшим коэффициентом.

Случайный поиск и везение

Решение поисковой задачи методом полного перебора (описанным ранее в этой главе в блоке «Избегаем решений методом полного перебора») возможно. Преимущество такого подхода в том, что для использования этих алгоритмов не требуются специальные знания предметной области. Алгоритм полного перебора обычно использует максимально простой подход к решению задачи. Недостаток в том, что метод полного перебора эффективен только при небольшом количестве узлов. Вот некоторые распространенные алгоритмы поиска методом полного перебора:

Метод	Описание	Недостатки	Преимущества
Поиск в ширину	Начинает с корневого узла: сначала исследует все дочерние узлы, затем переходит на следующий уровень. Продвигается уровень за уровнем, пока не найдет решение	Требует хранить каждый узел в памяти, что означает значительное использование памяти при большом количестве узлов	Может проверять повторяющиеся узлы для экономии времени и всегда находит решение
Поиск в глубину	Начинает с корневого узла и исследует связанный набор дочерних узлов, пока не достигнет листового узла. Продвигается ветка за веткой, пока не найдет решение	Не может проверять повторяющиеся узлы, что способно привести к многократному прохождению одних и тех же путей	Эффективен в использовании памяти
Двухнаправленный поиск	Выполняет поиск одновременно от корневого узла и целевого узла, пока два пути поиска не встретятся посередине	Эффективен по времени и использует память эффективнее других подходов, всегда находит решение	Сложность реализации, требующая больше времени на разработку

Использование эвристики и функции стоимости

Для некоторых людей слово *эвристика* звучит слишком сложно. Было бы проще сказать, что алгоритм делает обоснованное предположение и пробует снова в случае неудачи. В отличие от методов полного перебора, эвристические алгоритмы учатся, итеративно пытаясь улучшить решение со временем. Они также используют функции стоимости для принятия более удачных решений. Следовательно, эвристические алгоритмы сложнее, но имеют явное преимущество при решении комплексных задач. Как и в случае с алгоритмами полного перебора, есть много эвристических алгоритмов, каждый со своим набором преимуществ, недостатков и особых требований. Ниже описаны некоторые из наиболее распространенных эвристических алгоритмов.

- » **Чистый эвристический поиск.** Раскрывает узлы в порядке их стоимости. Поддерживает два списка. Закрытый список содержит уже исследованные узлы, а открытый список – узлы, которые еще предстоит изучить. На каждой итерации алгоритм расширяет узел с наименьшей возможной стоимостью. Все его дочерние узлы помещаются в закрытый список, и вычисляются индивидуальные стоимости дочерних узлов. Алгоритм возвращает дочерние узлы с низкой стоимостью обратно в открытый список и удаляет дочерние узлы с высокой стоимостью.
 - **Поиск A*.** Отслеживает стоимость узлов по мере их исследования (выбирая наименее затратные) с помощью следующего уравнения:
$$f(n) = g(n) + h(n), \text{ где:}$$
 - n – идентификатор узла;
 - $g(n)$ – стоимость достижения узла на данное время;
 - $h(n)$ – предполагаемая стоимость достижения цели из данного узла;
 - $f(n)$ – предполагаемая стоимость пути от n до цели.
- » **Жадный поиск по первому наилучшему совпадению.** Выбирает путь, который ближе всего к цели, используя уравнение $f(n) = h(n)$. Он может находить решения довольно быстро, но может и заикливаться, поэтому многие не считают его оптимальным подходом для поиска решения.

§ 5. Оценка алгоритмов

Понимание того, как именно работают алгоритмы, важно, поскольку иначе невозможно определить, действительно ли алгоритм работает так, как вам нужно. Кроме того, без качественных измерений нельзя провести точные сравнения, чтобы понять, действительно ли нужно искать новый метод решения проблемы, когда старое решение работает слишком медленно или использует слишком много ресурсов. Знание основ для сравнения различных решений и выбора между ними — важнейший навык при работе с алгоритмами.

Проблема эффективности была частью открытия и разработки новых алгоритмов с тех пор, как появилась сама концепция алгоритмов, именно поэтому мы видим так много различных алгоритмов, конкурирующих в решении одной и той же задачи. Концепция измерения размера функций внутри алгоритма и анализа его работы не нова: еще в 1843 году Ада Лавлейс и Чарльз Бэббидж рассматривали проблемы эффективности алгоритмов применительно к компьютерам (см. на <https://www.computerhistory.org/babbage/adalovelace>).

Дональд Кнут (<https://www-cs-faculty.stanford.edu/~knuth>) — специалист по информатике, математик, почетный профессор Стэнфордского университета и автор этапного многотомного труда «*Искусство программирования*» (издательство Addison-Wesley) — посвятил значительную часть своих исследований сравнению алгоритмов. Он стремился формализовать способы оценки потребности алгоритмов в ресурсах математическим путем и обеспечить корректное сравнение альтернативных решений. А также ввел термин *анализ алгоритмов* — раздел информатики, посвященный формальному пониманию работы алгоритмов. Анализ измеряет необходимые ресурсы с точки зрения количества операций, требуемых алгоритму для получения решения, или занимаемого алгоритмом пространства (например, объема памяти компьютера, необходимого алгоритму).

Анализ алгоритмов требует математического понимания и вычислений, однако он чрезвычайно полезен в вашем путешествии по изучению, оценке и эффективному использованию алгоритмов. Эта тема значительно абстрактнее, чем другие темы в книге. Чтобы сделать обсуждение менее теоретическим, в последующих главах представлены более практические аспекты такого измерения путем детального рассмотрения алгоритмов. В следующих блоках вы познакомитесь с основами.

Моделирование с использованием абстрактных машин

Чем больше операций требует алгоритм, тем он сложнее. Сложность — это мера эффективности алгоритма с точки зрения затрат времени, поскольку каждая операция занимает определенное время. При решении одной и той же задачи сложные алгоритмы обычно менее предпочтительны, чем простые, поскольку требуют больше времени. Подумайте о тех случаях, когда скорость выполнения имеет решающее значение, например в медицинском или финансовом секторе либо при полете на автопилоте в самолете или космической ракете. Измерение сложности алгоритма — непростая задача, хотя и необходимая, если хотите применить правильное решение. Первый метод измерения использует абстрактные машины, такие как машина с произвольным доступом (Random Access Machine, RAM).

Абстрактные машины — это не реальные компьютеры, а теоретические, то есть компьютеры, работа которых существует только в воображении. Это что-то вроде мечтаний для специалистов по информатике. Вы используете абстрактные машины, чтобы оценить, насколько хорошо алгоритм будет работать на компьютере, не тестируя его на реальном устройстве, но с учетом ограничений используемого оборудования. RAM-компьютер выполняет базовые арифметические операции и взаимодействует с информацией в памяти, и это все. Каждый раз, когда RAM-компьютер что-то делает, это занимает один временной шаг (единицу времени). При оценке алгоритма в RAM-симуляции вы подсчитываете временные шаги, используя следующую процедуру:

- 1** считаете каждую простую операцию (арифметическую) как один временной шаг;
- 2** разбиваете сложные операции на простые арифметические операции и подсчитываете временные шаги, как определено в шаге 1;
- 3** считаете каждое обращение к данным в памяти как один временной шаг.

Для выполнения такого подсчета вы записываете псевдокод вашего алгоритма (как упоминалось в главе 1) и выполняете эти шаги с помощью бумаги и карандаша. В итоге это простой подход, основанный на базовом представлении о работе компьютеров, полезное приближение, которое можно использовать для сравнения решений независимо от мощности и скорости вашего оборудования или используемого языка программирования.



Использование симуляции отличается от запуска алгоритма на компьютере, поскольку вы используете стандартные и предопределенные входные данные. Реальные компьютерные измерения требуют запуска кода и проверки времени, необходимого для его выполнения. Запуск кода на компьютере фактически является *эталонным тестированием* — другой формой измерения эффективности, при которой вы также учитываете среду выполнения (например, тип используемого оборудования и программную реализацию). Эталонное тестирование полезно, но ему не хватает обобщения. Подумайте, например, как новое оборудование может быстро выполнить алгоритм, который раньше занимал вечность на вашем предыдущем компьютере.

Еще более абстрактный подход

Если предыдущие блоки показались абстрактными, то на фоне этой главы предыдущие разделы покажутся максимально конкретными. Соберитесь с силами и двигайтесь дальше, потому что вы точно справитесь с этой задачей! Измерение последовательности шагов, разработанных для решения проблемы, создает немало сложностей. В предыдущем блоке обсуждался подсчет временных шагов (количество операций), но иногда вам нужно учитывать и пространство (например, память, потребляемую алгоритмом). Пространство учитывается, когда ваша задача требовательна к ресурсам. В зависимости от проблемы алгоритм может считаться лучшим, когда он эффективно работает с точки зрения одного из следующих аспектов потребления ресурсов:

- » времени выполнения,
- » требований к оперативной памяти,
- » использования жесткого диска,
- » энергопотребления,
- » скорости передачи данных в сети.

Некоторые из этих аспектов связаны с другими обратной зависимостью, поэтому, если, например, вы хотите увеличить скорость выполнения, иногда можно увеличить потребление памяти или энергии, чтобы этого достичь. Вы можете не только иметь различные конфигурации эффективности при выполнении алгоритма, но и изменять характеристики оборудования и программную реализацию для достижения своих целей. С точки зрения оборудования использование суперкомпьютера или компьютера общего назначения имеет значение, а программное обеспечение или язык, используемый для написания алгоритма, определено

меняет правила игры. Кроме того, количество и тип данных, которые вы подаете алгоритму, могут привести к лучшим или худшим показателям производительности.

Симуляции на машине с произвольным доступом (RAM) учитывают время, потому что, когда вы можете применить решение в столь многих средах, а его использование ресурсов зависит от множества факторов, вы должны найти способ упростить сравнения, чтобы они стали стандартными. В противном случае вы не сможете сравнивать возможные альтернативы. Решение, как часто бывает со многими другими проблемами, заключается в использовании единой меры, подходящей для всех случаев. В данном случае мерой служит время, которое приравнивается к количеству операций, то есть сложности алгоритма.

Моделирование на машине с произвольным доступом (RAM) помещает алгоритм в ситуацию, которая не зависит как от языка программирования, так и от типа компьютера. Однако объяснение принципов работы RAM-моделирования другим требует значительных усилий. При анализе алгоритмов предлагается взять количество операций, полученное в результате RAM-моделирования, и преобразовать его в математическую функцию, которая описывает поведение алгоритма с точки зрения времени, то есть количества шагов или операций, необходимых при увеличении объема входных данных. Например, если ваш алгоритм сортирует объекты, вы можете выразить его сложность через функцию, показывающую, сколько операций требуется в зависимости от количества обрабатываемых объектов.

Работа с функциями

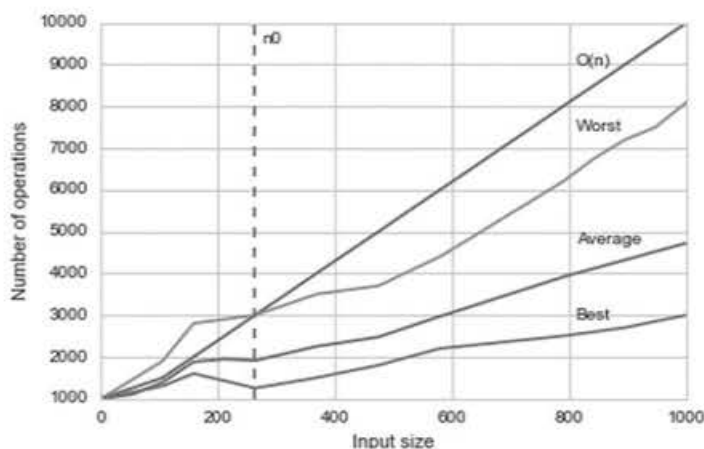
Функция в математике — это просто способ отображения некоторых входных данных в результат. Другими словами, *функция* — это преобразование (основанное на математических операциях), которое трансформирует (отображает) ваши входные данные в ответ. Для определенных значений входных данных (обычно обозначаемых буквами x или n) вы получаете соответствующий ответ, используя математику, определяющую функцию. Например, функция $f(n) = 2n$ говорит, что, когда на входе число n , ответом будет это число n , умноженное на 2.

Функция, описывающая связь между решением алгоритма и количеством получаемых им данных, может анализироваться без привязки к конкретному оборудованию или программному обеспечению. Ее легко сравнивать и с другими решениями при заданном размере задачи. Анализ алгоритмов действительно потрясающая концепция, поскольку сводит сложную последовательность шагов к математической формуле.

В большинстве случаев при анализе алгоритмов не требуется точного определения функции. Что действительно нужно, так это сравнение целевой функции с одной или несколькими другими функциями. Функции для сравнения входят в набор предложенных функций, которые работают хуже в сравнении с целевой функцией. Таким образом, вам не нужно подставлять числа в функции большей или меньшей сложности; вместо этого вы имеете дело с простыми, готовыми и хорошо известными функциями. Это эффективнее и похоже на классификацию производительности алгоритмов по категориям, а не на получение точного измерения производительности.

Этот набор обобщенных функций называется *Big-O-нотацией*, и в книге вы часто будете встречать небольшой набор функций (заклученных в скобки и предваряемых заглавной буквой *O*), используемых для представления производительности алгоритмов. На рисунке 2.1 показан анализ алгоритма. Его функцию, измеренную с помощью RAM-моделирования, можно представить в декартовой системе координат, где *абсцисса* (координата *x*) — это размер входных данных, а *ордината* (координата *y*) — результирующее количество операций. На графике представлены три кривые. Размер входных данных имеет значение. Однако качество тоже важно (например, при решении задач сортировки быстрее упорядочить входные данные, которые уже почти упорядочены). Следовательно, анализ показывает худший случай $f1(n)$, средний случай $f2(n)$ и лучший случай $f3(n)$. Хотя средний случай может дать общее представление, на самом деле важнее всего худший случай, поскольку проблемы могут возникнуть именно тогда, когда алгоритм с трудом находит решение. Функция *Big O* — это та функция, которая после определенного значения n_0 (порога для рассмотрения входных данных как больших) всегда дает большее количество операций при тех же входных данных, чем функция худшего случая $f1$. Таким образом, функция *Big O* даже более пессимистична, чем функция, представляющая ваш алгоритм, поэтому независимо от качества входных данных вы можете быть уверены, что хуже уже не будет.

РИСУНОК 2.1.
Сложность алгоритма для лучшего, среднего и худшего случая входных данных



Есть много функций, которые могут давать худшие результаты, но выбор функций, предлагаемых *Big-O*-нотацией, ограничен, поскольку ее цель — упростить измерение сложности путем предложения стандарта. Этот блок содержит лишь несколько функций, которые являются частью *Big-O*-нотации. Ниже они описаны в порядке возрастания сложности.

- » **Константная сложность $O(1)$.** Одинаковое время выполнения независимо от объема входных данных. В конечном счете это постоянное число операций, независимо от размера входных данных.
- » **Логарифмическая сложность $O(\log n)$.** Количество операций растет медленнее, чем входные данные, делая алгоритм менее эффективным на малых входных данных и более эффективным — на больших. Типичный алгоритм этого класса — бинарный поиск, описанный в главе 7, посвященной организации и поиску данных.
- » **Линейная сложность $O(n)$.** Операции растут вместе с входными данными в соотношении 1: 1. Типичный алгоритм — итерация, когда вы однократно просматриваете входные данные и применяете операцию к каждому их элементу. Глава 4 рассматривает итерации.
- » **Линейно-логарифмическая сложность $O(n \log n)$.** Сложность представляет собой сочетание логарифмической и линейной сложности. Она характерна для некоторых эффективных алгоритмов сортировки, таких как сортировка слиянием, пирамидальная сортировка и быстрая сортировка. Большинство из них описаны в главе 7.
- » **Квадратичная сложность $O(n^2)$.** Операции растут как квадрат количества входных данных. Когда одна итерация вложена в другую итерацию, вы получаете квадратичную сложность. Например, если у вас есть список имен и для поиска наиболее похожих вы сравниваете каждое имя со всеми остальными именами. Некоторые менее эффективные алгоритмы сортировки имеют такую сложность: сортировка пузырьком, сортировка выбором и сортировка вставками.
- » **Кубическая сложность $O(n^3)$.** Количество операций растет еще быстрее, чем при квадратичной сложности, из-за наличия нескольких вложенных итераций. Когда алгоритм имеет такой порядок сложности и обрабатывает умеренный объем данных (100 000 элементов), его выполнение

может занять годы. Когда количество операций является степенью входных данных, принято говорить, что алгоритм работает за полиномиальное время.

- » **Экспоненциальная сложность $O(2^n)$.** Алгоритм требует вдвое больше операций в сравнении с предыдущим шагом для каждого нового добавленного элемента. Когда у алгоритма такая сложность, даже небольшие задачи могут выполняться бесконечно долго. Многие алгоритмы полного перебора обладают экспоненциальной сложностью. Однако классический пример такого уровня сложности – вычисление чисел Фибоначчи (этот рекурсивный алгоритм рассматривается в главе 4).
- » **Факториальная сложность $O(n!)$.** Если на входе 100 объектов, а одна операция на компьютере занимает 10 секунд (разумная скорость для современных компьютеров), то для выполнения задачи потребуется около 10^1 лет (невозможное количество времени, учитывая, что возраст Вселенной оценивается в 10^1 лет). Известная задача с факториальной сложностью – задача коммивояжера, в которой требуется найти кратчайший маршрут для посещения множества городов с возвращением в исходный город (рассматривается в главе 18).

В ЭТОЙ ГЛАВЕ

- » Рассмотрение возможностей, предоставляемых Google Colab
- » Использование Google Colab для выполнения распространенных задач разработки
- » Запуск приложений в Google Colab
- » Получение помощи, когда она необходима

ГЛАВА 3

Работа с Google Colab

Сolaboratory (<https://colab.research.google.com/notebooks/welcome.ipynb>), или просто Colab, — это облачный сервис Google, который позволяет писать код на Python в среде, похожей на блокнот, в отличие от обычной интегрированной среды разработки. (Jupyter Notebook, <https://jupyter.org>, предоставляет аналогичную Colab-среду для настольных компьютеров, если у вас нет подключения к интернету.) Для его использования не нужно ничего устанавливать на свою систему. Преимущество такого подхода в том, что вы можете работать с кодом небольшими фрагментами и получать практически мгновенные результаты. Формат блокнота удобен и для создания отчетов, которые хорошо подходят для презентаций. В первом параграфе главы рассматриваются основы работы с Colab и объясняется, чем Colab отличается от стандартной среды разработки (и почему это различие дает существенные преимущества при изучении алгоритмов).

Вы можете использовать Colab для выполнения определенных задач в парадигме, ориентированной на ячейки. В следующих параграфах главы рассматриваются темы, связанные с решением задач, начиная с использования блокнотов. Конечно, вы захотите выполнять и другие виды задач, такие как создание различных типов ячеек и использование их для создания блокнотов, которые имеют вид отчета с работающим кодом.

Часть работы с Colab заключается в том, чтобы знать, как запускать примеры кода и делать это максимально быстро. Два параграфа главы посвящены использованию аппаратного ускорения и различным способам запуска примеров кода.

Наконец, поскольку глава не может охватить все аспекты Colab, последний параграф служит удобным ресурсом для поиска наиболее надежной информации о Colab.



ЗАПОМНИТЕ

Вам не нужно вводить исходный код этой главы вручную. На самом деле, использовать загружаемый исходный код намного проще. Исходный код главы можно найти в файле `\A4D2E\A4D2E; 03: Colab Examples.ipynb` из загружаемых исходников. Подробнее о том, как найти эти исходные файлы, смотрите во введении.

§ 1. Определение Google Colab

Colab разработан как аналог настольного приложения Jupyter Notebook (<https://jupyter.org>). Фактически довольно сложно различить функциональность этих двух приложений. Google Colab — это облачная версия Notebook, и это очевидно уже на странице приветствия. Он даже использует файлы IPython (предыдущее название Jupyter) Notebook (`.ipynb`) для сайта.

Хотя оба приложения похожи и используют файлы `.ipynb`, между ними есть некоторые различия, о которых вам нужно знать. В предыдущем издании книги использовался Jupyter Notebook, но поскольку Colab предоставляет возможность выполнять вычисления где угодно на любом устройстве с браузером, это издание фокусируется на Colab. Следующие блоки помогут вам понять различия в Colab.

Понимание функций Google Colab

С помощью Colab можно выполнять много задач, но для целей книги вы будете использовать его, чтобы написать и запустить код, создать соответствующую документацию и отобразить графику. Загружаемый исходный код для книги разработан, чтобы работать в Colab, но при желании вы можете использовать его и с Jupyter Notebook.

Jupyter Notebook — это локальное приложение, в котором вы используете локальные ресурсы. Потенциально вы могли бы использовать и другие источники, но в некоторых случаях это может оказаться неудобным или невозможным. Например, согласно <https://docs.github.com/repositories/working-with-files/using-files/working-with-non-code-files>, ваши файлы Notebook будут отображаться как статические HTML-страницы при использовании репозитория GitHub (*GitHub* — это облачная технология хранения, специально ориентированная на работу с кодом). Фактически некоторые функции вообще не будут работать. Colab позволяет полноценно взаимодействовать с вашими файлами Notebook, используя GitHub в качестве репозитория, и поддерживает другие варианты онлайн-хранения, поэтому Colab можно рассматривать в качестве вашего онлайн-партнера в создании кода на Python.

Другая причина, почему вам действительно нужно знать о Colab, заключается в том, что вы можете использовать его на альтернативном устройстве. В процессе написания некоторые примеры кода тестировались на планшете на базе Android (ASUS ZenPad 3S 10). На целевом планшете установлен Chrome, и он достаточно хорошо выполняет код для работы с примерами. При этом вы, вероятно, не захотите писать код на планшете такого размера: текст был невероятно мелким, и отсутствие клавиатуры тоже может стать проблемой. Суть в том, что для работы с кодом не обязательно, чтобы у вас была система Windows, Linux или OS X, но альтернативные варианты могут не обеспечить той производительности, которой вы ожидаете.



ЗАПОМНИТЕ

Google Colab обычно работает только с браузером Chrome (который используется в этой главе), Firefox или Safari (первые тесты с Microsoft Edge тоже показали обнадеживающие результаты). В большинстве случаев при попытке запустить Colab в неподдерживаемом браузере вы увидите сообщение об ошибке вроде «Этот сайт может не работать в вашем браузере. Пожалуйста, используйте поддерживаемый браузер» и ничего более. Ссылка More Info («Подробнее») ведет на <https://research.google.com/colaboratory/faq.html#browsers>, где вы можете получить дополнительную информацию.

НЕКОТОРЫЕ ОСОБЕННОСТИ FIREFOX

Даже при наличии онлайн-справки в вашей версии Firefox может появляться сообщение об ошибке **SecurityError: The operation is insecure** («Операция небезопасна»). В первоначальном диалоговом окне ошибки указывается на не связанную с проблемой причину, например файлы cookie, но это сообщение об ошибке появляется, когда вы нажимаете

Details («Подробности»). Если просто закрыть диалоговое окно, нажав «ОК», может показаться, что Colab работает, поскольку он отображает ваш код, но результатов выполнения кода вы не увидите.

Первым шагом к исправлению этой проблемы будет убедиться, что у вас установлена актуальная версия Firefox, поскольку старые версии не обеспечивают необходимой поддержки. После обновления браузера установка параметра `network.websocket.allowInsecureFromHTTPS` в значение `True` через **About: Config** должна решить проблему, но иногда этого недостаточно. В таком случае убедитесь, что Firefox действительно разрешает сторонние cookie, выбрав опцию Always («Всегда») для параметра Accept Third Party Cookies and Site Data («Принимать сторонние cookie и данные сайтов») и опцию Remember History («Запоминать историю») в разделе History («История») на вкладке Privacy & Security («Приватность и защита») в диалоговом окне настроек. Перезапускайте Firefox после каждого изменения и попробуйте запустить Colab снова. Если ни одно из этих исправлений не поможет, вам придется использовать Chrome для работы с Colab на вашей системе.

Знакомство с возможностями Google Colab

Google Colab предоставляет доступ к ряду функций через систему меню. Одна из этих функций — аппаратное ускорение — рассматривается в параграфе «Использование аппаратного ускорения» далее в этой главе. Функции, описанные в этом блоке, доступны в меню Tools («Инструменты»).

Поиск команд

При выборе пункта меню Tools ⇨ Command Palette («Инструменты ⇨ Палитра команд») отображается список команд, которые можно выполнить, как показано на рисунке 3.1. У некоторых из этих команд также есть сочетания клавиш, например Ctrl + Alt + M для добавления комментария к ячейке. Все эти команды помогают выполнять задачи, связанные с содержимым блокнота, — такие, как добавление форм.

РИСУНОК 3.1.
Использование
команд Colab
облегчает на-
стройку вашего
блокнота



Настройка параметров

Опция Tools ⇒ Settings («Инструменты ⇒ Настройки») открывает диалоговое окно настроек, показанное на рисунке 3.2. Четыре вкладки настроек выполняют следующие задачи:

- » **Site («Сайт»).** Настраивает работу сайта. Наиболее интересный параметр – тема оформления. Выбор Adaptive («Адаптивная») позволяет Colab подбирать цвета интерфейса в зависимости от условий освещения. На этой вкладке также можно настроить параметры отображения и доступа.
- » **Editor («Редактор»).** Определяет, как текст отображается на экране и как работает интерфейс. Например, при желании можно настроить привязки клавиш так, чтобы они работали как в *Vim* (текстовый редактор, включенный в системы Unix и Linux, обычно как утилита *vi*; см. на <https://www.vim.org>). Можно выбрать также размер шрифта, количество пробелов для каждого уровня отступа и много других настроек.
- » **Colab Pro.** Содержит рекламу Colab Pro (<https://colab.research.google.com/signup>), который дает значительные преимущества, такие как более быстрые графические процессоры, увеличенное время выполнения и больше памяти, – все это позволяет выполнять больше работы за меньшее время.

- » **Miscellaneous («Разное»)**. Содержит забавные настройки. Можно выбрать три визуальных эффекта с настройкой уровня мощности: позволить корги, котенку или крабу бегать по верхней части экрана. Можно выбрать любую комбинацию этих эффектов.

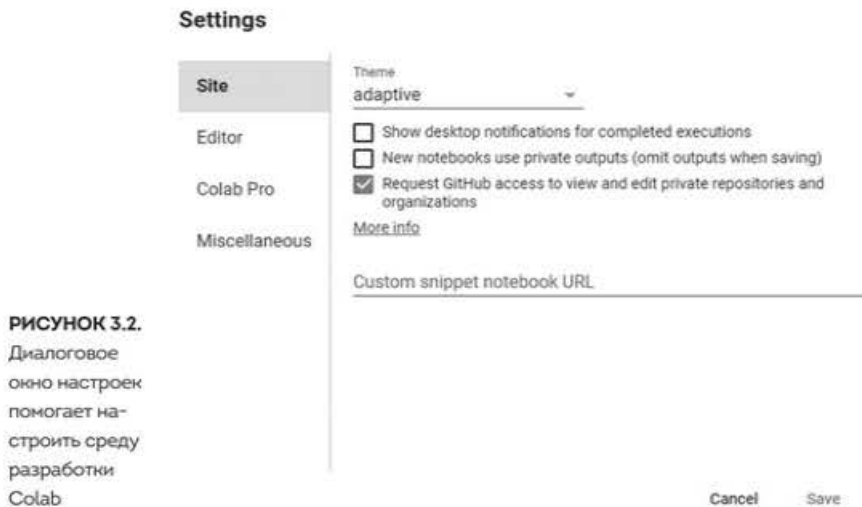


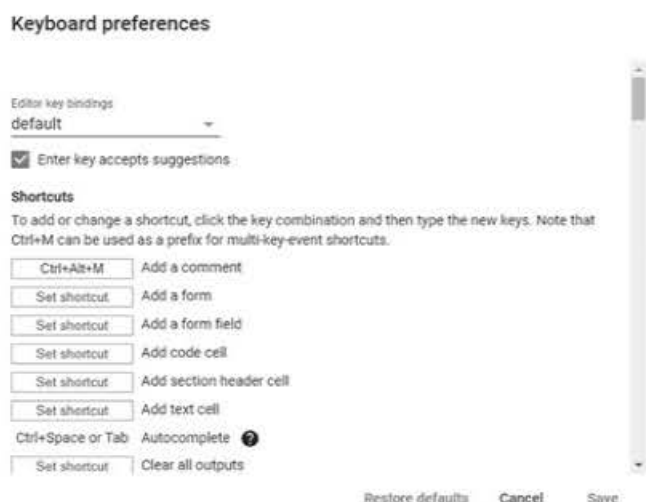
РИСУНОК 3.2.
Диалоговое
окно настроек
помогает на-
строить среду
разработки
Colab

Настройка сочетаний клавиш

Если вам не нравятся сочетания клавиш по умолчанию, вы можете настроить их в соответствии со своими потребностями. Для этого выберите Tools ⇒ Keyboard Shortcuts («Инструменты ⇒ Сочетания клавиш»), и вы увидите диалоговое окно настроек клавиатуры, показанное на рисунке 3.3. Если видите Set Shortcut («Установить сочетание клавиш»), это означает, что для команды в настоящее время нет сочетания клавиш, поэтому при желании вы можете его добавить. Вот как работать с сочетаниями клавиш:

- » чтобы добавить или изменить сочетание клавиш, поместите курсор в поле рядом с командой и нажмите комбинацию клавиш, которую хотите использовать для этой команды;
- » чтобы удалить ярлык, нажмите Delete («Удалить»).

РИСУНОК 3.3.
Настройте сочетания клавиш для быстрого доступа к командам



Сравнение файлов

Иногда нужно сравнить два файла, чтобы увидеть, чем они различаются. Когда вы выбираете Tools ⇒ Diff Notebooks («Инструменты ⇒ Сравнение блокнотов»), Colab открывает новую вкладку браузера и показывает два блокнота рядом, как показано на рисунке 3.4. Это файлы, случайным образом выбранные из вашего Google Drive. Чтобы выбрать файлы, с которыми вы действительно хотите работать, нажмите стрелку вниз рядом с путем к файлу в каждой панели. Различия отображаются на экране.

РИСУНОК 3.4.
Colab позволяет сравнивать два файла, чтобы увидеть их различия



§ 2. Работа с блокнотами

Блокнот — основа для взаимодействия с Colab. Фактически Colab построен на блокнотах, как упоминалось ранее. Когда вы наводите курсор на определенные части страницы приветствия по адресу <https://colab.research.google.com/notebooks/welcome.ipynb>, вы видите возможности для взаимодействия со страницей путем добавления кода или текстовых записей (которые при необходимости можно использовать для заметок). Эти записи активны, поэтому вы можете с ними взаимодействовать. Вы также можете перемещать ячейки и копировать полученный материал на свой Google Drive. Конечно, хотя взаимодействие со страницей приветствия является одновременно неожиданным и увлекательным, основная цель главы — показать, как взаимодействовать с блокнотами Colab. В следующих блоках описывается, как выполнять базовые задачи, связанные с блокнотами в Colab.

Создание нового блокнота

Чтобы создать новый блокнот, выберите File ⇒ New Notebook («Файл ⇒ Новый блокнот»). Вы увидите новый блокнот Python 3, как показано на рисунке 3.5. (Последняя версия Colab не поддерживает Python 2, поэтому вы не можете его использовать.)

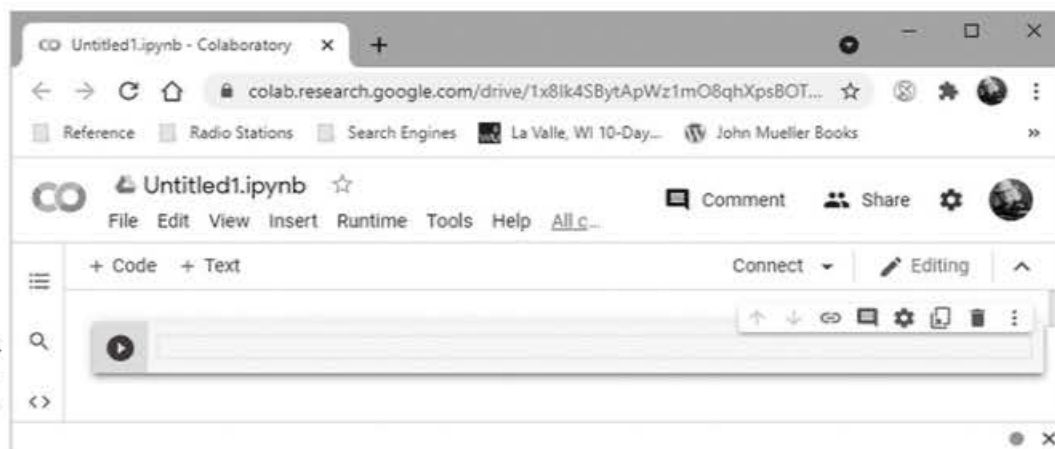


РИСУНОК 3.5.
Создание нового блокнота Python 3

Блокнот, показанный на рисунке 3.5, позволяет изменить имя файла, щелкнув по нему. Чтобы выполнить код в определенной ячейке, нужно нажать на стрелку, направленную вправо, с левой стороны этой ячейки. После выполнения кода нужно напрямую выбрать следующую ячейку.

Открытие существующих блокнотов

Вы можете открывать существующие блокноты, находящиеся в локальном хранилище, на Google Drive или GitHub. Доступны также примеры Colab и возможность загрузки файлов из других источников, например с сетевого диска вашей системы. В любом случае начните с выбора пункта меню File ⇒ Open Notebook («Файл ⇒ Открыть блокнот»). Появится диалоговое окно, показанное на рисунке 3.6.

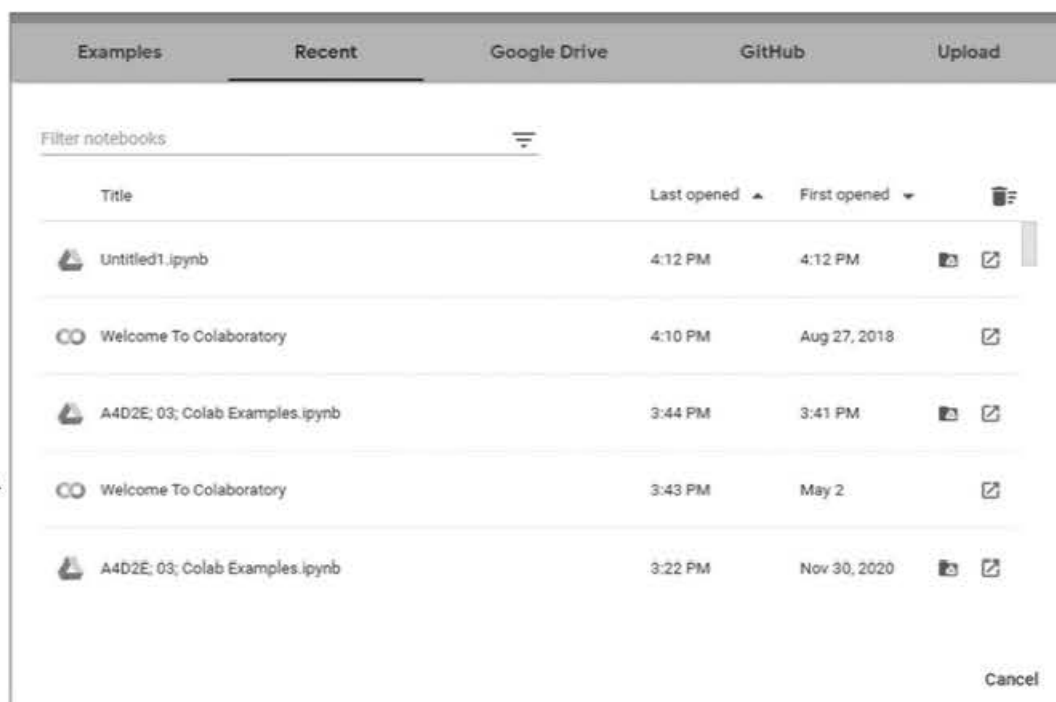


РИСУНОК 3.6. Используйте это диалоговое окно для открытия существующих блокнотов

По умолчанию отображаются все недавно открытые файлы, независимо от их расположения. Файлы представлены в алфавитном порядке. Вы можете отфильтровать список, введя строку в поле Filter Notebooks («Фильтровать блокноты»). В верхней части окна находятся дополнительные варианты открытия блокнотов.



СОВЕТ

Даже если вы не авторизованы, у вас все равно есть доступ к примерам проектов Colab. Эти проекты помогут вам разобраться с Colab, хотя и не позволят работать с собственными проектами. Тем не менее вы можете экспериментировать с Colab без предварительной авторизации в Google. В следующих блоках подробно рассматриваются доступные варианты.

Использование Google Drive для существующих блокнотов

Google Drive — место по умолчанию для многих операций в Colab, и вы всегда можете выбрать его в качестве места назначения. При работе с Google Drive вы увидите список файлов, аналогичный показанному на рисунке 3.6. Чтобы открыть конкретный файл, щелкните по его ссылке в диалоговом окне. Файл откроется в текущей вкладке браузера.

Использование GitHub для существующих блокнотов

При работе с GitHub сначала нужно указать расположение исходного кода онлайн, как показано на рисунке 3.7. Ссылка должна вести на публичный проект; использовать Colab для доступа к частным проектам нельзя.

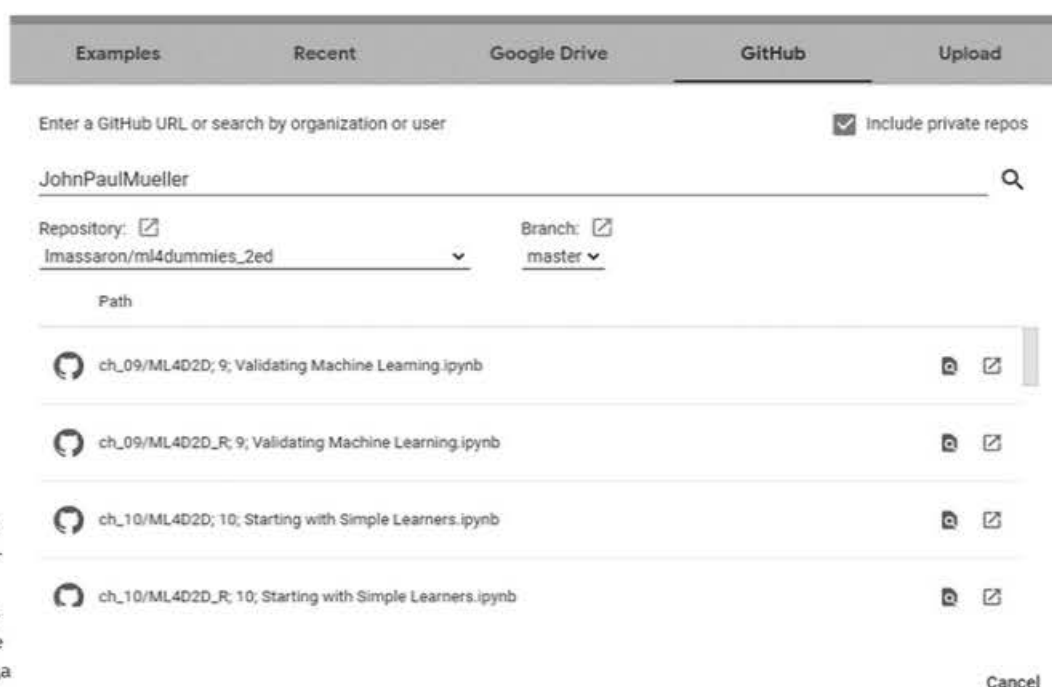


РИСУНОК 3.7. При использовании GitHub нужно указать расположение исходного кода

После подключения к GitHub вы увидите два списка: репозитории, которые служат контейнерами для кода, относящегося к определенному проекту, и ветки — конкретные реализации кода. При выборе репозитория и ветки отображается список файлов блокнотов, которые можно загрузить в Colab. Просто нажмите на нужную ссылку, и файл загрузится так же, как если бы вы использовали *Google Drive* (<https://drive.google.com>) — еще один тип онлайн-хранилища.

Использование локального хранилища для существующих блокнотов

Если хотите использовать исходные файлы книги или любые другие локальные файлы, выберите вкладку Upload («Загрузить») в диалоговом окне. В центре находится единственная кнопка Choose File («Выбрать файл»). При нажатии на нее открывается диалоговое окно выбора файла вашего браузера. Найдите нужный файл для загрузки, как вы обычно это делаете при открытии любого файла.



ЗАПОМНИТЕ

После выбора файла и нажатия кнопки Open («Открыть») файл загружается на Google Drive. Если вы вносите изменения в файл, они сохраняются на Google Drive, а не на вашем локальном диске. В зависимости от браузера обычно открывается новое окно с загруженным кодом. Однако вы можете увидеть только сообщение об успешной загрузке — в этом случае нужно открыть файл так же, как при использовании Google Drive. В некоторых случаях браузер спрашивает, хотите ли вы покинуть текущую страницу. Следует ответить утвердительно.



СОВЕТ

Команда File ⇒ Upload Notebook («Файл ⇒ Загрузить блокнот») также загружает файл на Google Drive. По сути, загрузка блокнота работает так же, как загрузка любого другого типа файлов, и вы увидите то же диалоговое окно. Если хотите загрузить файлы других типов, использование команды «Файл ⇒ Загрузить блокнот» может быть быстрее.

Сохранение блокнотов

Colab предоставляет много вариантов сохранения блокнота. Однако ни один из них не работает с вашим локальным диском. После загрузки контента с локального диска на Google Drive или GitHub, Colab управляет этим контентом в облаке, а не на вашем локальном диске. Чтобы сохранить обновления на локальном диске, вы должны скачать файл. В следующих блоках рассматриваются облачные варианты сохранения блокнотов.

Использование Google Drive для сохранения блокнотов

По умолчанию данные хранятся на Google Drive. Когда вы выбираете File Save («Файл ⇒ Сохранить»), созданный контент сохраняется в корневой каталог вашего Google Drive. Если хотите сохранить контент в другую папку, нужно выбрать эту папку на Google Drive (<https://drive.google.com>).



ЗАПОМНИТЕ

Colab отслеживает версии вашего проекта при каждом сохранении. Однако по мере устаревания этих версий Colab удаляет их. Чтобы сохранить версию, которая не будет удалена со временем, используйте команду File ⇒ Save and Pin Revision («Файл ⇒ Сохранить и закрепить версию»). Чтобы просмотреть версии вашего проекта, выберите File ⇒ Revision History («Файл ⇒ История версий»).

Вы также можете сохранить копию проекта, выбрав File ⇒ Save a Copy In Drive («Файл ⇒ Сохранить копию на Drive»). К имени копии добавляется слово *Copy* («Копия»). Разумеется, позже вы можете переименовать ее. Colab сохраняет копию в текущей папке Google Drive.

Использование GitHub для сохранения блокнотов

GitHub предоставляет альтернативу Google Drive для сохранения контента. Он предлагает организованный способ обмена кодом для обсуждения, проверки и распространения. GitHub доступен по адресу <https://github.com>. Исходный код этой книги находится по адресу https://github.com/lmassaron/algo4d_2ed, поэтому вы можете легко получить к нему доступ из Colab.



ЗАПОМНИТЕ

При работе с GitHub из Colab вы можете использовать только публичные репозитории, хотя GitHub поддерживает и частные репозитории. Чтобы сохранить файл на GitHub, выберите File ⇒ Save a Copy in GitHub («Файл ⇒ Сохранить копию на GitHub»). Если вы еще не вошли в GitHub, Colab отобразит окно с запросом данных для входа. После входа вы увидите диалоговое окно, похожее на показанное на рисунке 3.8.

Copy to GitHub

Repository: ☒	Branch: ☒
JohnPaulMueller/A4D2E	main
File path	
A4D2E; 03; Colab Examples.ipynb	
Commit message	
Created using Colaboratory	
☐ Include a link to Colaboratory	
Cancel OK	

РИСУНОК 3.8. Использование GitHub означает хранение данных в репозитории

Если у вашей учетной записи еще нет репозитория, вы должны либо создать новый репозиторий, либо выбрать существующий для хранения данных. После сохранения файл появится в выбранном репозитории GitHub. По умолчанию репозиторий включает ссылку для открытия данных в Colab, если вы не отключите эту функцию.

Использование GitHub gists для сохранения блокнотов

GitHub gists используются как способ обмена отдельными файлами или другими ресурсами с другими людьми. Некоторые используют их и для полноценных проектов, но основная идея в том, что у вас есть концепция, которой вы хотите поделиться, — что-то не совсем оформленное и не представляющее собой готовое приложение. Подробнее о gists можно прочитать на <https://help.github.com/articles/about-gists>.

§ 3. Выполнение общих задач

Большинство задач в Colab и Jupiter Notebook работают одинаково. И там, и там есть ячейки с кодом и без кода, и вы можете создавать ячейки с кодом как в Colab, так и в Notebook, используя опции в меню Insert («Вставка»). Аналогично в обеих средах есть ячейки без кода трех типов:

- » текст;
- » заголовок раздела;
- » поле формы, которое бывает следующих типов:
 - выпадающим списком,
 - полем ввода,
 - ползунком,
 - Markdown.

Ячейки без кода в Colab работают несколько иначе, чем ячейки Markdown в Notebook, но идея та же. Два интересных дополнения в Colab, которых нет в Notepad, — это черновая ячейка кода, позволяющая экспериментировать с кодом в реальном времени, и фрагменты кода — готовые блоки кода для выполнения определенных задач (их можно просто вставлять там, где необходимо).

Вы также можете редактировать и перемещать ячейки. Одно важное различие между этими двумя средами заключается в том, что в Colab нельзя изменить тип ячейки, а в Notebook — можно. Ячейка, созданная как заголовок раздела, не может внезапно превратиться в ячейку с кодом. В следующих блоках представлен краткий обзор различных функций.

Создание ячеек с кодом

Первая ячейка, которую создает Colab, — это ячейка с кодом. У Colab и Notebook есть одинаковые функции в отношении кода, поэтому код, написанный в Colab, работает и в Notebook (и наоборот). Однако сбоку от ячейки вы видите меню дополнительных функций, которые можно использовать в Colab (см. рисунок 3.9); в Notebook их нет.

РИСУНОК 3.9.

Ячейки с кодом в Colab содержат несколько дополнительных функций, которых нет в Notebook



Значки, показанные на рисунке 3.9, используются для расширения возможностей работы с кодом в Colab. Вот что делают эти функции (в порядке появления слева направо на рисунке):

- » **Переместить ячейку вверх.** Перемещает ячейку на одну позицию вверх в иерархии ячеек.
- » **Переместить ячейку вниз.** Перемещает ячейку на одну позицию вниз в иерархии ячеек.
- » **Ссылка на ячейку.** Отображает диалоговое окно со ссылкой, которую можно использовать для доступа к определенной ячейке в блокноте. Вы можете встроить эту ссылку в любое место на веб-странице или в блокноте, чтобы кто-то мог получить доступ к конкретной ячейке. Человек по-прежнему видит весь блокнот, но ему не нужно искать ячейку, которую вы хотите обсудить.
- » (опционально) **Добавить комментарий** (при наличии соответствующих прав). Создает облачко комментария справа от ячейки. Это не то же самое, что комментарий в коде, который существует внутри кода, но относится ко всей ячейке. Вы можете редактировать, удалять или помечать комментарии как решенные. Решенный комментарий — тот, который получил внимание и больше не актуален.
- » **Открыть настройки редактора.** Открывает то же диалоговое окно, что показано на рисунке 3.2 и обсуждается в блоке «Знакомство с возможностями Google Colab» ранее в этой главе. Этот параметр появляется только после выделения некоторого кода.

» **Отразить ячейку во вкладке.** Отражает выбранную ячейку в панели Cell («Ячейка»), которая появляется в правой части окна. Вы можете прокручивать код в левой панели где угодно и сохранять доступ к этому коду. Стрелка вправо позволяет выполнить ячейку в любое время после внесения изменений в код в левой панели. Пара двойных стрелок позволяет одним щелчком вернуться к выделенному коду в левой панели. Вы также можете переместить код ячейки в черновую ячейку, где можете экспериментировать с ним, не изменяя исходного кода. У вас может быть несколько панелей Cell. Просто выберите нужную и переключайтесь между ними по мере необходимости, что позволит легко перемещаться по коду. Закрыть панель Cell можно, нажав X рядом со словом *Cell*.

» **Удалить ячейку.** Удаляет ячейку из блокнота.

» **Вертикальное многоточие.** Содержит несколько дополнительных функций в меню (не все из которых могут отображаться, так как они зависят от открытого файла, выполненных задач и ваших прав на работу с содержимым файла):

- **Копировать ячейку** – копирует содержимое текущей выбранной ячейки в буфер обмена;
- **Вырезать ячейку** – удаляет содержимое текущей выбранной ячейки и помещает его в буфер обмена;
- **Очистить вывод** – удаляет вывод из ячейки (чтобы заново сгенерировать вывод, нужно повторно запустить код);
- **Просмотр вывода в полноэкранном режиме** – отображает вывод (не всю ячейку или другие части блокнота) в полноэкранном режиме на устройстве (эта опция полезна при отображении большого объема контента или когда детальный просмотр графики помогает объяснить тему: нажмите Esc для выхода из полноэкранного режима);
- **Добавить форму** – вставляет форму в ячейку справа от кода. (Формы используются для создания графического интерфейса ввода параметров. Формы не отображаются в Notebook, но благодаря способу их создания они не мешают выполнению кода в Notebook. Подробнее о формах можно прочитать на <https://colab.research.google.com/notebooks/forms.ipynb>.)

Ячейки кода также предоставляют информацию о коде и его выполнении. Маленький значок рядом с выводом показывает информацию о выполнении при наведении на него курсора мыши. Щелчок по значку очищает вывод. Чтобы заново сгенерировать вывод, нужно повторно запустить код.

Создание текстовых ячеек

Текстовые ячейки работают аналогично ячейкам разметки в Notebook. Однако, как показано на рисунке 3.10, вы получаете дополнительную помощь в форматировании текста через графический интерфейс. Разметка остается той же, но у вас есть возможность использовать графический интерфейс для ее создания. Например, в данном случае, чтобы создать символ решетки (#) для заголовка, нужно нажать на значок с двойной буквой Т, который появляется первым в списке. Повторное нажатие на значок двойной Т увеличит уровень заголовка. Справа вы увидите, как текст будет выглядеть в блокноте.

Обратите внимание на меню справа от текстовой ячейки. Оно содержит многие из тех же опций, что и ячейка кода. Например, вы можете создать список ссылок, чтобы помочь людям получить доступ к определенным частям вашего блокнота через оглавление. В отличие от Notebook, вы не можете выполнять текстовые ячейки для обработки содержащейся в них разметки.

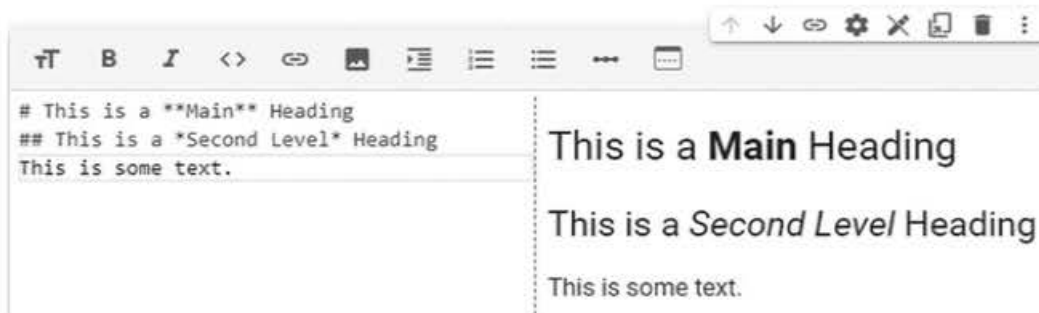
Создание специальных ячеек

Специальные ячейки, предоставляемые Colab, — это разновидности текстовых ячеек. Эти особые ячейки, доступ к которым осуществляется через меню Insert («Вставка»), позволяют быстрее создавать необходимые ячейки. Однако не стоит использовать специальные ячейки, если вам нужно сохранить совместимость между Colab и Notebook. В следующих блоках описаны все типы специальных ячеек.

Работа с заголовками

При выборе Insert ⇒ Section Header Cell («Вставка ⇒ Ячейка заголовка раздела») под текущей выбранной ячейкой создается новая ячейка с соответствующим заголовком первого уровня. Вы можете увеличить уровень заголовка, нажав на значок с двойной буквой Т. Графический интерфейс выглядит так же, как показано на рисунке 3.10, поэтому у вас есть доступ ко всем стандартным функциям форматирования текста.

РИСУНОК 3.10.
Используйте
графический
интерфейс для
упрощения
форматирова-
ния текста



Работа с оглавлением

Интересное дополнение в Colab — автоматическое создание оглавления для вашего блокнота. Чтобы воспользоваться этой функцией, нажмите на значок Table of Contents («Оглавление») в левой части окна. Оглавление содержит по одной записи для каждого заголовка в вашем коде. Записи автоматически организуются по уровням, так что вы видите иерархию своего кода. При нажатии на запись вы автоматически переходите к соответствующему месту в коде.

Редактирование ячеек

И в Colab, и в Notebook есть меню Edit («Правка») с ожидаемыми опциями, такими как вырезать, копировать и вставлять ячейки. Между двумя продуктами есть и некоторые интересные различия. Например, Notebook позволяет разделять и объединять ячейки. Colab содержит опцию для показа или скрытия кода в виде переключателя. Эти различия придают каждому продукту свой особый характер, но не меняют вашу способность использовать их для создания и модификации кода на Python.

Перемещение ячеек

Тот же метод, который вы используете для перемещения ячеек в Notebook, работает и в Colab. Единственное различие в том, что Colab полагается исключительно на кнопки панели инструментов (см. рисунок 3.9); в Notebook также есть опции перемещения ячеек в меню Edit («Правка»). Чтобы переместить ячейку, выберите ее и нажимайте кнопки Move cell up («Переместить ячейку вверх») или Move cell down («Переместить ячейку вниз») по необходимости.

§ 4. Использование аппаратного ускорения

Хотя это не понадобится для примеров в этой книге, Colab предлагает аппаратное ускорение в виде графического процессора (ГПУ) или тензорного процессора (ТПУ). Оба этих специальных процессора обеспечивают возможность параллельной обработки множества наборов данных на высокой скорости. При работе с большими данными (см. главу 12) в среде машинного или глубокого обучения ГПУ или ТПУ может значительно сократить время, необходимое для выполнения задачи. Основное различие между ГПУ и ТПУ заключается в том, что ГПУ присутствует в составе большинства современных высокопроизводительных видеоадаптеров и может использоваться для рендеринга сложной графики, в то время как ТПУ — это процессор, разработанный Google специально для задач машинного и глубокого обучения. (Существуют и другие различия, но они не важны для этой книги.)

По умолчанию поддержка ГПУ и ТПУ в Colab отключена. Чтобы включить поддержку ГПУ или ТПУ, выберите Runtime ⇒ Change Runtime Type («Среда выполнения ⇒ Изменить тип среды выполнения»). Появится диалоговое окно Notebook Settings («Настройки блокнота»). В этом окне находится выпадающий список Hardware Accelerator («Аппаратный ускоритель»), в котором можно выбрать None («Никакой») (по умолчанию), GPU («ГПУ») или TPU («ТПУ»).

§ 5. Выполнение кода

Чтобы код был полезен, его нужно когда-то запустить. В предыдущих блоках упоминалась направленная вправо стрелка, которая появляется в текущей ячейке. Нажатие на стрелку запускает только текущую ячейку. Конечно, помимо нажатия на стрелку, есть и другие варианты, и все они доступны в меню Runtime («Среда выполнения») (в Notebook это меню Cell [«Ячейка»]). Вот краткий обзор этих опций:

- » **Запуск текущей ячейки.** Вместо нажатия на стрелку вправо можно также выбрать Runtime ⇒ Run the Focused Cell («Среда выполнения ⇒ Выполнить код в сфокусированной ячейке»), чтобы выполнить код в текущей ячейке.

- » **Запуск других ячеек.** Colab предоставляет опции в меню «Среда выполнения» для выполнения кода в следующей ячейке, предыдущей ячейке или группе выбранных ячеек. Просто выберите опцию, соответствующую ячейке или набору ячеек, которые вы хотите выполнить.
- » **Запуск всех ячеек.** В некоторых случаях вам может понадобиться выполнить весь код в блокноте. В этом случае выберите Runtime ⇒ Run All («Среда выполнения ⇒ Выполнить все»). Выполнение начнется сверху блокнота, с первой ячейки, содержащей код, и продолжится до последней ячейки с кодом в блокноте. Вы можете остановить выполнение в любую минуту, выбрав Runtime ⇒ Interrupt Execution («Среда выполнения ⇒ Прервать выполнение кода»).



СОВЕТ

При выборе Runtime ⇒ Manage Sessions («Среда выполнения ⇒ Управление сеансами») отображается диалоговое окно со списком всех сеансов, которые в данную минуту выполняются для вашей учетной записи в Colab. С помощью этого диалогового окна можно определить, когда код в этом блокноте выполнялся в последний раз и сколько памяти потребляет блокнот. Нажмите на значок корзины, чтобы прекратить выполнение конкретного блокнота.

§ 6. Получение помощи

Самое очевидное место для получения помощи по Colab — это меню Help («Помощь») в Colab. В меню нет общей ссылки на справку, но вы можете найти ее по адресу <https://colab.research.google.com/notebooks/welcome.ipynb> (для этого нужно войти на сайт Colab). Это меню содержит все обычные пункты:

- » **Часто задаваемые вопросы (FAQ).** Переход на страницу с вопросами, которые задавали другие пользователи.
- » **Поиск фрагментов кода.** Открывает панель, показывающую типовые задачи, например работу с камерой, где вы можете искать примеры кода, которые могли бы соответствовать вашим потребностям после небольшой модификации. Нажатие кнопки Insert («Вставка») вставляет код в текущую позицию курсора в активной ячейке. Каждая запись также содержит пример кода.

- » **Сообщить об ошибке.** Открывает страницу, где можно сообщить об ошибках в Colab.
- » **Задать вопрос на Stack Overflow.** Открывает новую вкладку браузера, где можно задать вопросы другим пользователям. Если вы еще не вошли в Stack Overflow, появится экран входа.
- » **Отправить отзыв.** Отображает диалоговое окно со ссылками на дополнительную информацию. Если вы действительно хотите отправить отзыв, нажмите ссылку Continue Anyway («Все равно продолжить») внизу диалогового окна.

В ЭТОЙ ГЛАВЕ

- » Использование матриц и векторов для выполнения вычислений
- » Получение правильных комбинаций
- » Применение рекурсивных методов для получения определенных результатов
- » Рассмотрение способов ускорения вычислений

ГЛАВА 4

Выполнение базовых операций с данными на Python

Вероятно, вы уже изучили основы языка Python с помощью онлайн-уроков или других методов — те загадочные символы, которые используются для общения с компьютером. (Если нет, хорошие базовые уроки можно найти на <https://www.w3schools.com/python> и <https://www.tutorialspoint.com/python/index.htm>.) Однако просто знать, как управлять языком, используя его конструкции для выполнения задач, недостаточно для создания полезного приложения. Цель математических алгоритмов — преобразовать один вид данных в другой. Манипулирование данными означает взять необработанные входные данные и сделать с ними что-то для достижения желаемого результата. (Эта тема рассматривается в книге «*Python для анализа данных для чайников*» Джона Пола Мюллера и Луки Массарона [издательство Wiley]). Например, пока вы не обработаете данные о трафике, вы не увидите закономерностей, которые подскажут, куда вложить дополнительные средства для улучшений. Необработанные данные о трафике сами по себе ни о чем не говорят — их нужно обработать, чтобы увидеть закономерности в полезном виде.

В прошлом люди вручную выполняли различные манипуляции, чтобы сделать данные полезными, что требовало глубоких знаний математики. К счастью, сейчас можно найти пакеты Python для выполнения большинства этих манипуляций с помощью небольшого кода. Вам больше

не нужно запоминать загадочные манипуляции — достаточно знать, какие функции Python использовать. Именно в этом вам поможет данная глава. Вы узнаете способы выполнения различных видов манипуляций с данными с помощью легкодоступных пакетов Python, специально разработанных для этой цели. (Глава 5 рассмотрим как создать собственную библиотеку алгоритмов, написанных вручную.) Эта глава начинается с операций над векторами и матрицами. В последующих параграфах рассматриваются такие методы, как рекурсия, способные сделать задачи еще проще, а также выполнить некоторые задачи, которые практически невозможно решить другими способами. Вы также узнаете, как ускорить вычисления, чтобы тратить меньше времени на манипуляции с данными и больше времени — на что-то действительно интересное.



ЗАПОМНИТЕ

Вам не нужно вводить исходный код для этой главы вручную. На самом деле, использовать загружаемый исходный код намного проще. Исходный код для этой главы можно найти в файлах `\A4D2E\A4D2E; 04; Basic Vectors and Matrixes.ipynb`, `\A4D2E\A4D2E; 04; Binary Search.ipynb` и `\A4D2E\A4D2E; 04; Recursion.ipynb` из загружаемых исходников. Подробнее о том, как найти эти исходные файлы, смотрите во введении.

§ 1. Выполнение вычислений с использованием векторов и матриц

Для выполнения полезной работы в Python часто требуется работать с большими объемами данных, которые представлены в определенных формах. У этих форм странно звучащие названия, но они весьма важны. Вот три термина, которые вам нужно знать для этой главы:

- » **скаляр** – единичный базовый элемент данных (например, число 2, представленное само по себе, является скаляром);
- » **вектор** – одномерный массив (по сути, список) элементов данных (например, массив, содержащий числа 2, 3, 4 и 5, будет вектором);
- » **матрица** – двух- или многомерный массив (по сути, таблица) элементов данных (например, массив, содержащий числа 2, 3, 4 и 5 в первой строке и 6, 7, 8 и 9 во второй строке, является матрицей).

Python сам по себе предоставляет интересный набор возможностей, но для выполнения некоторых задач все равно потребовалось бы много работы. Чтобы сократить объем работы, можно положиться на код, написанный другими людьми и собранный в пакеты. В следующих блоках описывается, как использовать пакет NumPy (<https://numpy.org>) для выполнения различных операций над скалярами, векторами и матрицами. Эта глава представляет обзор NumPy, уделяя особое внимание функциям, которые вы будете использовать позже (подробнее см. на <https://www.w3schools.com/python/numpy/default.asp>).

Понимание скалярных и векторных операций

Пакет NumPy предоставляет основные функциональные возможности для научных вычислений в Python. Чтобы использовать `numpy`, вы импортируете его с помощью команды вроде `import numpy as np`. Теперь вы можете обращаться к `numpy`, используя общепринятое двухбуквенное сокращение `np`.



ЗАПОМНИТЕ

Python предоставляет доступ только к одному типу данных в каждой конкретной категории. Например, если вам нужно создать переменную, представляющую число без десятичной части, вы используете целочисленный тип данных. Использование такого общего обозначения полезно, поскольку упрощает код и уменьшает количество деталей, о которых должен беспокоиться разработчик. Однако в научных расчетах часто требуется лучший контроль над тем, как данные представлены в памяти, что означает необходимость в большем количестве типов данных, которые `numpy` вам предоставляет. Например, вам может потребоваться определить конкретный скаляр как `short` (значение длиной 16 бит). Используя `numpy`, вы можете определить переменную как `myShort = np.short(15)`. Пакет NumPy предоставляет доступ к набору различных типов данных (<https://numpy.org/doc/stable/reference/arrays.scalars.html>).

Для создания вектора используйте функцию `numpy array()`. Например, `myVect = np.array([1, 2, 3, 4])` создает вектор с четырьмя элементами. В данном случае вектор содержит стандартные целые числа Python. Вы также можете использовать функцию `arange()` для создания векторов, например: `myVect = np.arange(1, 10, 2)`, которая заполнит `myVect` значениями `[1, 3, 5, 7, 9]`. Первый аргумент задает начальную точку, второй — конечную точку, а третий — шаг между числами. Четвертый аргумент позволяет определить тип данных для вектора.

Вы также можете создать вектор с определенным типом данных. Для этого достаточно указать тип данных следующим образом: `myVect =`

`np.int16([1, 2, 3, 4])` — это заполнит `myVect` вектором, содержащим 16-битные целочисленные значения. Чтобы проверить это самостоятельно, используйте `print(type(myVect[0]))`, что выведет `<class 'numpy.int16'>`.



СОВЕТ

Вы можете выполнять базовые математические операции над векторами целиком, что делает `numpy` невероятно полезным и менее подверженным ошибкам, которые могут возникнуть при использовании таких программных конструкций, как циклы, для выполнения той же задачи. Например, если начать с `myVect = np.array([1, 2, 3, 4])`, то `myVect + 1` даст результат `array([2, 3, 4, 5], dtype=int16)`. Обратите внимание, что в выводе явно указывается используемый тип данных. Как можно предположить, `myVect - 1` даст результат `array([0, 1, 2, 3], dtype=int16)`.

В завершение разговора о скалярных и векторных операциях стоит отметить, что вы также можете выполнять логические операции и операции сравнения. Например, следующий код выполняет операции сравнения двух массивов:

```
a = np.array([1, 2, 3, 4])
b = np.array([2, 2, 4, 4])

print(a == b)
print(a < b)
```

Результат в этом случае:

```
[False True False True]
[ True False True False]
```

Начиная с двух векторов, `a` и `b`, код проверяет, равны ли отдельные элементы в `a` соответствующим элементам в `b`. В данном случае `a[0]` не равно `b[0]`. Однако `a[1]` равно `b[1]`. Результатом является вектор типа `bool`, содержащий значения `True` или `False` на основе отдельных сравнений.

Логические операции используют специальные функции. Вы можете проверить логический вывод булевых операторов AND, OR, XOR и NOT. Вот пример логических функций:

```
a = np.array([True, False, True, False])
b = np.array([True, True, False, False])

print(np.logical_or(a, b))
print(np.logical_and(a, b))
print(np.logical_not(a))
print(np.logical_xor(a, b))
```

При запуске этого кода вы увидите следующие результаты:

```
[True True True False]
[True False False False]
[False True False True]
[False True True False]
```

Подробнее о логических функциях можно прочитать на странице <https://numpy.org/doc/stable/reference/routines.logic.html>.

Выполнение умножения векторов

Сложение, вычитание или деление векторов происходит поэлементно, как описано в предыдущем блоке. Однако когда дело доходит до умножения, все становится немного странным. На самом деле, в зависимости от того, что вы действительно хотите сделать, ситуация может стать весьма необычной. Рассмотрим тип умножения, описанный в предыдущем блоке. И `myVect * myVect`, и `np.multiply(myVect, myVect)` дают поэлементный результат `[1, 4, 9, 16]` при работе с массивом `[1, 2, 3, 4]`.



ВНИМАНИЕ

К сожалению, поэлементное умножение может давать неверные результаты при работе с алгоритмами. Во многих случаях вам действительно нужно *скалярное произведение* — сумма произведений двух числовых последовательностей. При работе с векторами скалярное произведение всегда является суммой отдельных поэлементных умножений и дает в результате одно число. Например, `myVect.dot(myVect)` дает результат `30`. Если просуммировать значения из поэлементного умножения, вы обнаружите, что они действительно дают в сумме `30`. Материал по адресу <https://www.mathsisfun.com/algebra/vectors-dot-product.html> рассказывает о скалярных произведениях и помогает понять, где они могут пригодиться в алгоритмах. Узнать больше о функциях линейной алгебры для `numpy` можно на странице <https://numpy.org/doc/stable/reference/routines.linalg.html>.

Создание матрицы – правильный способ начать

Многие приемы, которые вы используете с векторами, работают и с матрицами. Чтобы создать базовую матрицу, просто используйте функцию `array()`, как и в случае с вектором, но определите дополнительные измерения. *Измерение* — это направление в матрице. Например, двумерная матрица содержит строки (одно направление) и столбцы (второе направ-

ление). Вызов массива `myMatrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])` создает матрицу, содержащую три строки и три столбца:

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Обратите внимание, как три списка вложены во внешний список для создания двух измерений. Чтобы получить доступ к определенному элементу массива, укажите индексы строки и столбца, например `myMatrix[0, 0]` для доступа к первому значению 1. Полный список функций создания векторных и матричных массивов можно найти по адресу <https://numpy.org/doc/stable/reference/routines.array-creation.html>.



ЗАПОМНИТЕ

Пакет NumPy поддерживает специальный класс `matrix`. У этого класса особые возможности, которые упрощают выполнение специфических для матриц задач. Вы узнаете об этих возможностях позже в этой главе. Пока все, что вам действительно нужно знать, — это как создать матрицу типа данных `matrix`. Самый простой способ — сделать вызов, похожий на тот, что используется для функции `array`, но используя вместо нее функцию `mat`, например `myMatrix = np.mat([[1, 2, 3], [4, 5, 6], [7, 8, 9]])`:

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Чтобы убедиться, что это действительно матрица, попробуйте выполнить `print(type(myMatrix))`, что выведет `<class 'numpy.matrix'>`. Вы также можете преобразовать существующий массив в матрицу с помощью функции `asmatrix()`. Используйте функцию `asarray()` для преобразования объекта `matrix` обратно в форму `array`.



ВНИМАНИЕ

Единственный недостаток класса `matrix` в том, что он работает только с двумерными матрицами. При попытке преобразовать трехмерную матрицу в класс `matrix` вы увидите сообщение об ошибке, говорящее о том, что размерность слишком велика для матрицы.

Умножение матриц

При умножении двух матриц возникают те же вопросы, что и при умножении двух векторов (как обсуждалось в блоке «Выполнение умножения векторов» ранее в этой главе). Следующий код выполняет поэлементное умножение двух матриц:


```
a = np.array([[1,2,3],[4,5,6]])
b = np.array([[1,2,3],[4,5,6]])

print(a*b)
```

Вывод выглядит так:

```
[[1 4 9]
 [16 25 36]]
```



ВНИМАНИЕ

Обратите внимание, что **a** и **b** — одинаковой формы: две строки и три столбца. Для выполнения поэлементного умножения обе матрицы должны быть одинаковой формы. В противном случае вы увидите сообщение об ошибке, говорящее о несоответствии размерностей. Как и в случае с векторами, функция `multiply()` также производит поэлементный результат.

Скалярное произведение работает с матрицами иначе. В этом случае количество столбцов матрицы **a** должно совпадать с количеством строк матрицы **b**. При этом количество строк в матрице **a** может быть любым, как и количество столбцов в матрице **b**, если вы умножаете **a** на **b**. Например, следующий код производит корректное скалярное произведение:

```
a = np.array([[1,2,3],[4,5,6]])
b = np.array([[1,2,3],[3,4,5],[5,6,7]])

print(a.dot(b))
```

с выводом:

```
[[22 28 34]
 [49 64 79]]
```

Обратите внимание, что выходной массив содержит количество строк из матрицы **a** и количество столбцов из матрицы **b**. Как же это работает? Чтобы получить значение 22 в выходном массиве по индексу `[0, 0]`, нужно сложить значения $a[0, 0] * b[0, 0]$ (что равно 1), $a[0, 1] * b[1, 0]$ (что равно 6) и $a[0, 2] * b[2, 0]$ (что равно 15), получив в итоге 22. Остальные элементы вычисляются точно так же.



СОВЕТ

Для выполнения поэлементного умножения двух матричных объектов нужно использовать функцию `numpy.multiply()`.

Определение расширенных матричных операций

В книге рассматриваются различные интересные матричные операции, но некоторые из них используются часто, поэтому они появляются в этой главе. При работе с массивами иногда данные приходят в форме, которая не работает с алгоритмом. К счастью, `numpy` поставляется со специальной функцией `reshape()`, которая позволяет преобразовать данные в любую нужную форму. Фактически вы можете использовать ее для преобразования вектора в матрицу, как показано в следующем коде:

```
changeIt = np.array([1,2,3,4,5,6,7,8])
print(changeIt)

changeIt = changeIt.reshape(2,4)
print(changeIt)

changeIt = changeIt.reshape(2,2,2)
print(changeIt)
```

При запуске этого кода вы увидите следующие выводы (пробелы добавлены для ясности):

```
[1 2 3 4 5 6 7 8]

[[1 2 3 4]
 [5 6 7 8]]

[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
```



ЗАПОМНИТЕ

Исходная форма `changeIt` — это вектор, но использование функции `reshape()` превращает его в матрицу. Кроме того, вы можете придать матрице любое количество измерений, которое работает с данными. Однако вы должны задать форму, соответствующую требуемому количеству элементов. Например, вызов `changeIt.reshape(2, 3, 2)` завершится неудачей, поскольку элементов недостаточно для создания матрицы такого размера.

В некоторых алгоритмических формулировках вы можете встретить две важные матричные операции: транспонирование и обращение матрицы. *Транспонирование* происходит, когда матрица размерности $n \times m$ преобразуется в матрицу $m \times n$ путем замены строк на столбцы. В большинстве

текстов эта операция обозначается верхним индексом T , например A^T . Чаще всего эта операция используется при умножении для получения правильных размерностей. При работе с `numpy` для выполнения этой операции используется функция `transpose`. Например, начав с матрицы, имеющей две строки и четыре столбца, вы можете ее транспонировать, чтобы получить четыре строки по два столбца в каждой, как показано в этом примере:

```
changeIt = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
print(changeIt)

changeIt = np.transpose(changeIt)
print(changeIt)
```

Результаты выглядят так:

```
[[1 2 3 4]
 [5 6 7 8]]
[[1 5]
 [2 6]
 [3 7]
 [4 8]]
```

Обращение матрицы применяется к матрицам размерности $m \times m$ — квадратным матрицам, имеющим одинаковое количество строк и столбцов. Эта операция очень важна, поскольку позволяет непосредственно решать уравнения с матричным умножением, такие как $y = bX$, где известны вектор y и матрица X , а нужно найти значения вектора b . Так как для большинства скалярных чисел (за исключением нуля) существует число, умножение на которое дает 1, идея состоит в том, чтобы найти обратную матрицу, умножение на которую даст специальную матрицу, называемую *единичной матрицей*. Чтобы увидеть единичную матрицу в `numpy`, используйте функцию `identity`, например так:

```
print(np.identity(4))
```

что дает следующий результат:

```
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

Обратите внимание, что единичная матрица содержит все единицы по диагонали. Найти обратное число для скаляра довольно просто (для скалярного числа n обратным будет n^{-1} , то есть $1/n$). С матрицей дело

обстоит иначе. Обращение матрицы требует довольно большого количества вычислений. Обратная матрица для матрицы A обозначается как A^{-1} . При работе с `numpy` для создания обратной матрицы используется функция `linalg.inv()`. В следующем примере показано, как создать обратную матрицу, использовать ее для получения скалярного произведения, а затем сравнить это скалярное произведение с единичной матрицей с помощью функции `allclose()`.

```
a = np.array([[1,2], [3,4]])
b = np.linalg.inv(a)

print(np.allclose(np.dot(a, b), np.identity(2)))
```



ЗАПОМНИТЕ

Результат `True` говорит о том, что `b` является обратной матрицей для `a`. Иногда найти обратную матрицу невозможно. Когда матрицу нельзя обратить, ее называют *сингулярной матрицей*, или *вырожденной матрицей*. Сингулярные матрицы не являются нормой; они встречаются довольно редко.

§ 2. Создание комбинаций правильным способом

Формирование данных часто требует рассмотрения их с разных точек зрения. Данные — это не просто последовательность чисел, а значимая последовательность, которая при правильном упорядочивании передает информацию наблюдателю. Создание нужных комбинаций данных путем манипулирования последовательностями — важная часть настройки алгоритмов для выполнения требуемых задач. В следующих блоках рассматриваются три метода формирования данных: перестановки, комбинации и повторения.

Работа с перестановками

Когда вы получаете необработанные данные, они имеют определенный порядок. Он может отражать практически что угодно, например журнал устройства ввода данных, отслеживающего производственную линию. Возможно, это серия чисел, представляющих количество продукции, произведенной в конкретное время. Причина, по которой данные поступают в определенном порядке, важна, но порядок может не подходить

для получения нужного результата от алгоритма. Создание *перестановки* данных — переупорядочивание данных для получения другого представления — может помочь достичь желаемого результата.

На перестановки можно смотреть по-разному. Один из способов рассмотрения перестановки — как случайное представление порядка последовательности. В этом случае можно использовать функцию `numpy.random.permutation()`, как показано здесь:

```
a = np.array([1,2,3])
print(np.random.permutation(a))
```

То, что вы видите, — это рандомизированная версия исходных данных, например `[2 1 3]`. Каждый раз при запуске этого кода вы получаете другой случайный порядок последовательности данных, что очень удобно для алгоритмов, требующих рандомизации набора данных, чтобы получить желаемые результаты. Например, выборка — важная часть анализа данных, а показанный метод — эффективный способ выполнения этой задачи.

Другой способ рассмотреть эту задачу — необходимость получить все перестановки набора данных, чтобы попробовать каждую по очереди. Для выполнения этой задачи нужно импортировать пакет `itertools`. Следующий код показывает метод, который можно использовать для получения списка всех перестановок конкретного вектора:

```
from itertools import permutations

a = np.array([1,2,3])

for p in permutations(a):
    print(p)
```

Вывод выглядит так:

```
(1, 2, 3)
(1, 3, 2)
(2, 1, 3)
(2, 3, 1)
(3, 1, 2)
(3, 2, 1)
```

Если хотите использовать подход *списковых включений* (https://www.w3schools.com/python/python_lists_comprehension.asp), который является более коротким методом выполнения повторяющихся задач, то можете использовать `[print(p) for p in permutations(a)]` вместо этого. Подробнее об `itertools` можно прочитать на <https://docs.python.org/3/library/itertools.html>.

Перемешивание комбинаций

В некоторых случаях вам не нужен весь набор данных; все, что действительно нужно, — это несколько элементов в комбинациях определенной длины. Например, у вас может быть набор данных, содержащий четыре числа, и вам нужны только комбинации из двух чисел. (Возможность получать части набора данных — ключевая функция для генерации полносвязного графа, который описан в разделе 3 книги.) Следующий код показывает, как получить такие комбинации:

```
from itertools import combinations

a = np.array([1,2,3,4])

for comb in combinations(a, 2):
    print(comb)
```

что дает следующий вывод:

```
(1, 2)
(1, 3)
(1, 4)
(2, 3)
(2, 4)
(3, 4)
```

Вывод содержит все возможные двухчисленные комбинации `a`. Обратите внимание, что в этом примере используется функция `combinations()` из модуля `itertools` (функция `permutations()` рассматривалась в предыдущем блоке). Конечно, вам могут быть нужны не все эти комбинации; возможно, случайная выборка из них подойдет лучше. В этом случае вам поможет функция `random.sample()`, как показано здесь:

```
import random

pool = []

for comb in combinations(a, 2):
    pool.append(comb)

print(random.sample(pool, 3))
```

Конкретные комбинации, которые вы увидите в выводе, будут разными, например `[(1, 2), (2, 3), (1, 4)]`. Однако суть в том, что вы ограничили набор данных двумя способами. Вы не используете, во-первых, все элементы данных одновременно и, во-вторых, все возможные комбинации этих

элементов. В результате создается относительно случайно выглядящий набор элементов данных, который можно использовать как входные данные для алгоритма. Python предоставляет целый ряд методов рандомизации, с которыми можно ознакомиться на странице <https://docs.python.org/3/library/random.html>. Многие последующие примеры в книге также полагаются на рандомизацию для получения правильного результата работы алгоритмов.

Борьба с повторениями

Повторяющиеся данные могут несправедливо повлиять на результат работы алгоритма, что приведет к неточным результатам. Иногда для определения результата обработки данных нужны уникальные значения. К счастью, Python позволяет легко удалять определенные типы повторяющихся данных. Рассмотрим пример:

```
a = np.array([1,2,3,4,5,6,6,7,7,1,2,3])
b = np.array(list(set(a)))

print(b)
```

Вывод содержит только уникальные элементы:

```
[1, 2, 3, 4, 5, 6, 7]
```



ЗАПОМНИТЕ

В данном случае *a* начинается с набора чисел в произвольном порядке и со множеством повторений. В Python множество никогда не содержит повторяющихся данных. Следовательно, преобразовав список *a* во *множество*, затем обратно в *список* и поместив список в *массив*, вы получите вектор без повторений.

§ 3. Получение желаемых результатов с помощью рекурсии

Рекурсия — элегантный метод решения многих компьютерных задач, который основан на способности функции продолжать вызывать саму себя, пока не будет удовлетворено определенное условие. Сам термин «рекурсия» происходит от латинского глагола *recurrere*, что означает «бежать назад».

Когда используете рекурсию, вы решаете задачу путем многократного вызова одной и той же функции, но изменяя условия, при которых вы ее вызываете. Основная причина использования рекурсии заключается в том, что она обеспечивает более простой способ решения задач при работе с некоторыми алгоритмами, поскольку имитирует то, как человек решил бы эту задачу. К сожалению, рекурсия — непростой инструмент, поскольку требует определенных усилий, чтобы понять того, как построить рекурсивную процедуру, и может вызвать проблемы с нехваткой памяти на вашем компьютере, если не настроить некоторые параметры памяти. В следующих блоках подробно рассказывается о том, как работает рекурсия, и приводится пример использования рекурсии в Python.

Объяснение рекурсии

Многим людям сложно использовать рекурсию, потому что они не могут легко представить, как она работает. В большинстве случаев вы вызываете функцию Python, она что-то делает и останавливается. Однако при рекурсии вы вызываете функцию Python, она что-то делает, а затем многократно вызывает сама себя, пока задача не достигнет определенного условия, при этом все предыдущие вызовы остаются активными. Вызовы разворачиваются один за другим, пока первый вызов наконец не завершится с правильным ответом, и именно этот процесс разворачивания вызывает у большинства людей затруднения. На рисунке 4.1 показано, как выглядит рекурсия в виде блок-схемы.

Обратите внимание на условие в центре. Чтобы рекурсия работала, в функции должно быть такое условие, иначе она может превратиться в бесконечный цикл. Условие определяет одно из двух:

- » условия для завершения рекурсии не выполнены, поэтому функция должна вызвать себя снова;
- » условия для завершения рекурсии выполнены, поэтому функция возвращает конечное значение, которое используется для вычисления окончательного результата.

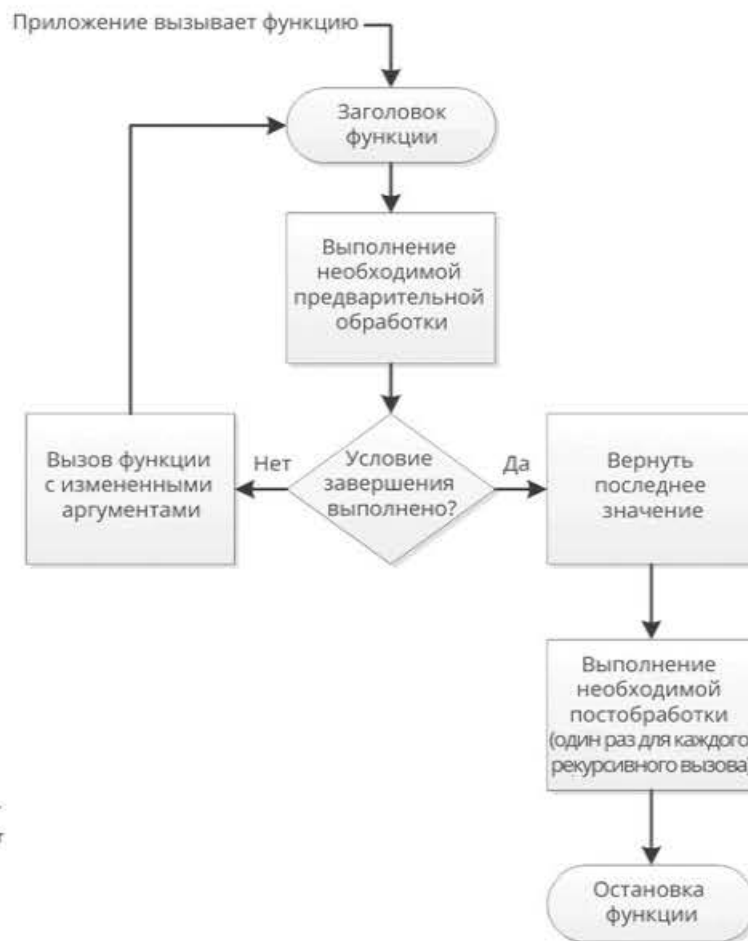


РИСУНОК 4.1.
В процессе рекурсии функция непрерывно вызывает себя, пока не выполнится определенное условие



ЗАПОМНИТЕ

Когда функция вызывает сама себя, она не использует те же аргументы, которые были ей переданы. Если бы она постоянно использовала одни и те же аргументы, условие никогда бы не изменилось и рекурсия никогда бы не закончилась. Следовательно, рекурсия требует, чтобы последующие вызовы функции изменяли аргументы вызова, чтобы приблизить функцию к конечному решению.

Один из самых распространенных примеров рекурсии для всех языков программирования — это вычисление факториала. *Факториал* — это умножение ряда чисел между начальной и конечной точкой, где каждое число в ряду на единицу меньше предыдущего. Например, чтобы вычислить $5!$ (читается как «пять факториал»), вы умножаете $5 \times 4 \times 3 \times 2 \times 1$. Это вычисление представляет собой идеальный и простой пример рекурсии. Вот код Python, который можно использовать для выполнения этого вычисления:

```
def factorial(n):
    print("factorial called with n = ", str(n))
```



```

if n == 1 or n == 0:
    print("Ending condition met.")
    return 1
else:
    return n * factorial(n-1)

print(factorial(5))

```

Вот результат, который вы увидите при запуске этого примера:

```

factorial called with n = 5
factorial called with n = 4
factorial called with n = 3
factorial called with n = 2
factorial called with n = 1
Ending condition met.
120

```

Код достигает конечного условия, когда $n == 1$. Каждый последующий вызов `factorial()` использует `factorial(n - 1)`, что уменьшает начальный аргумент на 1. Вывод показывает каждый последующий вызов факториала и достижение конечного условия. Результат, 120, равен 5! (пять факториал).

Важно понимать, что нет единственного способа использования рекурсии для решения задачи. Как и с любым другим приемом программирования, можно найти много способов достичь одного и того же результата. Например, вот другая версия рекурсивного вычисления факториала, которая использует меньше строк кода, но эффективно выполняет ту же задачу:

```

def factorial(n):
    print("factorial called with n = ", str(n))
    if n > 1:
        return n * factorial(n-1)
    print("Ending condition met.")
    return 1

print(factorial(5))

```



ЗАПОМНИТЕ

Обратите внимание на разницу. Вместо проверки условия завершения эта версия проверяет условие продолжения. Пока n больше 1, код будет продолжать делать рекурсивные вызовы. Хотя этот код короче предыдущей версии, он менее понятен, поскольку теперь вам нужно думать о том, какое условие прервет рекурсию.

Устранение хвостовой рекурсии

Многие формы рекурсии опираются на хвостовой вызов. Фактически первый пример в предыдущем блоке его использует. *Хвостовой вызов* происходит всякий раз, когда рекурсия вызывает функцию как последнее действие перед возвратом. В предыдущем блоке строка `return n * factorial(n - 1)` является хвостовым вызовом.

Хвостовые вызовы необязательно плохи, и они представляют собой способ, которым большинство людей пишут рекурсивные процедуры. Однако использование хвостового вызова заставляет Python отслеживать значения отдельных вызовов до тех пор, пока рекурсия не развернется обратно. Каждый вызов потребляет память. Когда-нибудь у системы закончится память, и вызов завершится неудачей, что приведет к сбою вашего алгоритма. С учетом сложности и огромных наборов данных, используемых некоторыми современными алгоритмами, хвостовые вызовы могут доставить немало хлопот тем, кто их использует.

С помощью небольшого программистского мастерства вы можете потенциально устранить хвостовые вызовы из ваших рекурсивных процедур. В интернете можно найти множество поистине удивительных приемов, таких как «использование трамплина», как объясняется на <https://blog.moertel.com/posts/2013-06-12-recursion-to-iteration-4-trampolines.html>. (Заметьте, это не тот трамплин, с которого прыгают в воду!) Однако самый простой подход, если есть желание устранить рекурсию, — это создать итеративную альтернативу, которая выполняет ту же задачу. Например, вот функция `factorial()`, которая использует итерацию вместо рекурсии, чтобы исключить потенциальные проблемы с памятью:

```
def factorial(n):
    print("factorial called with n = ", str(n))
    result = 1
    while n > 1:
        result = result * n
        n = n - 1
        print("Current value of n is ", str(n))
    print("Ending condition met.")
    return result

print(factorial(5))
```

Вывод выглядит очень похоже на предыдущий:

```
factorial called with n = 5
Current value of n is 4
Current value of n is 3
```

```
Current value of n is 2
Current value of n is 1
Ending condition met.
120
```

Базовый поток выполнения этой функции такой же, как у рекурсивной функции. Цикл `while` заменяет рекурсивный вызов, но вам все равно нужно проверять то же условие и продолжать цикл, пока данные не будут удовлетворять условию. Результат тот же самый. Однако в некоторых случаях замена рекурсии итерацией нетривиальна, как рассматривается в примере на <https://blog.moertel.com/posts/2013-06-03-recursion-to-iteration-3.html>.

§ 4. Более быстрое выполнение задач

Очевидно, что выполнение задач как можно быстрее — это всегда идеальный вариант. Однако нужно тщательно оценивать методы, которые вы используете для достижения этой эффективности. Пожертвовать небольшим объемом памяти ради ускорения выполнения задачи — отличное решение, если у вас есть свободная память. В последующих главах книги рассматриваются различные способы ускорения выполнения задач, но некоторые базовые приемы вы можете применять независимо от того, с каким алгоритмом работаете сейчас. В следующих блоках рассмотрим некоторые из этих приемов.

Рассматриваем метод «разделяй и властвуй»

Некоторые задачи кажутся непреодолимыми, когда вы только приступаете к их решению. Возьмем, к примеру, написание книги. Если рассматривать книгу целиком, то написание ее кажется непосильной задачей. Однако если разбить книгу на главы и сосредоточиться только на одной главе, задача становится более выполнимой. Конечно, полноценная глава тоже может казаться пугающей, поэтому вы разбиваете ее на текст с заголовками первого уровня, что делает задачу еще реальнее, хотя все еще недостаточно простой. Заголовки первого уровня могут содержать заголовки второго уровня и так далее, пока вы не разобьете задачу написания темы на короткие статьи настолько, насколько это возможно. Именно так работает метод «разделяй и властвуй». Вы разбиваете задачу на более мелкие подзадачи, пока не найдете такую, которую сможете решить без особых затруднений.

Компьютеры тоже могут использовать подход «разделяй и властвуй». Попытка решить огромную задачу с колоссальным набором данных может занять дни — если такая задача вообще выполнима. Однако, разбив большую задачу на более мелкие части, вы можете решить ее гораздо быстрее и с меньшими затратами ресурсов. Например, при поиске записи в базе данных нет необходимости просматривать всю базу, если она отсортирована. Допустим, вы ищете слово *Hello* в базе данных. Можно начать с разделения базы данных пополам (буквы от *A* до *M* и буквы от *N* до *Z*). Буква *H* в слове *Hello* стоит раньше *M* в алфавите, поэтому вы рассматриваете первую половину базы данных, а не вторую. Разделив оставшуюся половину еще раз (буквы от *A* до *G* и буквы от *H* до *M*), вы обнаруживаете, что вам нужна вторая половина остатка, которая теперь составляет лишь четверть всей базы данных. Дальнейшие деления в итоге помогают найти именно то, что вы ищете, с просмотром лишь небольшой части всей базы данных. Такой подход к поиску называется *бинарным поиском*. Задача сводится к выполнению следующих шагов:

- 1 разделить рассматриваемое содержимое пополам;
- 2 сравнить ключи содержимого с искомым термином;
- 3 выбрать половину, содержащую ключ;
- 4 повторять шаги 1–3, пока не найдется ключ.



ЗАПОМНИТЕ

Большинство задач, решаемых методом «разделяй и властвуй», следуют похожему подходу, хотя некоторые из них могут быть довольно запутанными. Например, вместо простого деления базы данных пополам в некоторых случаях вы можете разделить ее на три части. Однако цель во всех случаях одна и та же: разделить задачу на меньшую часть и определить, можно ли решить задачу, используя только эту часть как обобщенный случай. После того как вы найдете обобщенный случай, который умеете решать, вы сможете использовать этот фрагмент для решения любого другого фрагмента. Следующий код демонстрирует предельно простую версию бинарного поиска, предполагающую, что список отсортирован:

```
def search(searchList, key):
    mid = int(len(searchList) / 2)
    print("Searching midpoint at ", str(searchList[mid]))

    if mid == 0:
        print("Key Not Found!")
        return key

    elif key == searchList[mid]:
        print("Key Found!")
        return searchList[mid]
```

```

elif key > searchList[mid]:
    print("searchList now contains ",
          searchList[mid: len(searchList)])
    search(searchList[mid: len(searchList)], key)

else:
    print("searchList now contains ",
          searchList[0: mid])
    search(searchList[0: mid], key)
aList = list(range(1, 21))
search(aList, 5)

```

При запуске этого кода вы увидите следующий результат:

```

Searching midpoint at 11
searchList now contains [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Searching midpoint at 6
searchList now contains [1, 2, 3, 4, 5]
Searching midpoint at 3
searchList now contains [3, 4, 5]
Searching midpoint at 4
searchList now contains [4, 5]
Searching midpoint at 5
Key Found!

```

Этот рекурсивный подход к бинарному поиску начинается со списка `aList`, содержащего числа от 1 до 20. Он ищет значение 5 в списке `aList`. Каждая итерация рекурсии начинается с поиска средней точки списка, `mid`, а затем использует эту среднюю точку для определения следующего шага. Когда ключ совпадает со значением в средней точке, значение найдено в списке и рекурсия завершается.



ЗАПОМНИТЕ

Обратите внимание, что этот пример выполняет один из двух рекурсивных вызовов. Когда ключ больше значения в средней точке существующего списка, `searchList[mid]`, код снова вызывает функцию `search`, но уже только с правой половиной оставшегося списка. Другими словами, каждый вызов `search` использует только половину списка из предыдущего вызова. Когда ключ меньше или равен `searchList[mid]`, функция `search` получает левую половину существующего списка.



ВНИМАНИЕ

Список может не содержать искомого значения, поэтому вы всегда должны предусмотреть способ выхода из рекурсии, иначе *стек* (специальная область памяти, используемая для хранения информации о вызовах; подробнее см. на <https://www.geeksforgeeks.org/stack-vs-heap-memory-allocation>) переполнится, что приведет к сообщению об ошибке.

В данном случае выход происходит, когда `mid == 0`, что означает отсутствие списка `searchList` для поиска. Например, если вы измените `search(aList, 5)` на `search(aList, 22)`, то получите следующий результат:

```
Searching midpoint at 11
searchList now contains [11, 12, 13, 14, 15, 16, 17, 18,
 19, 20]
Searching midpoint at 16
searchList now contains [16, 17, 18, 19, 20]
Searching midpoint at 18
searchList now contains [18, 19, 20]
Searching midpoint at 19
searchList now contains [19, 20]
Searching midpoint at 20
searchList now contains [20]
Searching midpoint at 20
Key Not Found!
```

Обратите внимание и на то, что код проверяет условие выхода перед выполнением любой другой работы, чтобы случайно не вызвать ошибку из-за отсутствия содержимого в `searchList`. При работе с рекурсией вы должны действовать проактивно, иначе позже придется столкнуться с последствиями.

Различение разных возможных решений

Рекурсия служит частью многих различных алгоритмических решений в программировании, как вы увидите в следующих главах. На самом деле, во многих случаях трудно обойтись без рекурсии, поскольку итеративный подход оказывается неинтуитивным, громоздким и времязатратным. Однако вы можете создать несколько различных версий одного и того же решения, у каждой из которых будут свои характеристики, недостатки и достоинства.

В этой главе не рассматривается последовательный поиск, поскольку он обычно занимает больше времени, чем любое другое решение. В лучшем случае для завершения последовательного поиска требуется всего одно сравнение, но в худшем — искомый элемент находится при последней проверке. В среднем последовательный поиск требует $(n + 1) / 2$ проверок или времени $O(n)$ для завершения. (Подробнее о *Big-O*-нотации рассказывается в блоке «Работа с функциями» главы 2.)

Бинарный поиск, описанный в предыдущем блоке, работает гораздо эффективнее последовательного. Он выполняется за логарифмическое время или $O(\log n)$. В лучшем случае требуется только одна проверка, как и при последовательном поиске; но результаты примера показывают, что даже в худшем случае, когда значение вообще отсутствует в списке, требуется всего шесть проверок вместо 21 проверки, необходимой для последовательного поиска.



ВНИМАНИЕ

Однако если смотреть только на время выполнения, полученные данные могут ввести вас в заблуждение, заставив думать, что конкретное решение будет невероятно хорошо работать в вашем приложении, хотя на самом деле это не так. Нужно также учитывать тип обрабатываемых данных, сложность создания решения и много других факторов. Именно поэтому в последующих примерах книги еще рассматриваются преимущества и недостатки каждого подхода — скрытые опасности выбора решения, которое кажется многообещающим, но затем не дает желаемого результата.

В ЭТОЙ ГЛАВЕ

- » Определение причин для создания собственного класса
- » Составление базового класса
- » Выполнение стандартных вычислений с матрицами
- » Использование полученного класса

ГЛАВА 5

Разработка класса для матричных вычислений

Использование библиотек Python, несомненно, облегчает жизнь разработчика, поскольку они выполняют большую часть работы по написанию кода. А если код должным образом проверен, разработчикам приходится меньше беспокоиться об ошибках в коде. Поэтому может быть трудно представить сценарий, в котором вы не захотите использовать такую библиотеку, как NumPy (<https://numpy.org>), для выполнения матричных операций. В конце концов, она обычно входит в десятку лучших библиотек Python (см. на <https://towardsdatascience.com/best-python-libraries-for-every-python-developer-77daab4fa40e>). Хотя использование NumPy в большинстве случаев выгодно, в первом параграфе главы обсуждается, почему иногда от него стоит отказаться, и рассказывается о некоторых преимуществах создания собственного класса для выполнения хотя бы базовых задач по работе с матрицами.

В следующем параграфе главы рассматривается процесс создания базового класса для работы с матрицами, включая такие основные операции, как умножение одной матрицы на другую. Этот класс будет лишь примером, не для использования в реальных проектах. Цель состоит в том, чтобы помочь вам понять, что включает в себя создание такого класса. Реальному классу, вероятно, потребуется предоставить больше функциональности, созданный нами отлично послужит для демонстрационных целей.

После создания базового класса, вы начнете добавлять в него функции для манипулирования матрицами, которые он создает. Например, вам может понадобиться узнать, как преобразовать вашу матрицу в плоский вид для определенных типов вычислений, — и это цель третьего параграфа главы. Таким образом, вы сможете разобраться, что может потребоваться для создания собственных расширений базового класса.



ЗАПОМНИТЕ

Вам не нужно вводить исходный код для этой главы вручную. На самом деле, использовать загружаемый исходный код намного проще. Вы можете найти исходный код для этой главы в файле `\A4D2E\A4D2E; 05: Matrix Class.ipynb` из загружаемых исходников. Подробнее о том, как найти этот исходный файл, смотрите во введении.

§ 1. Избегание использования NumPy

NumPy — это феноменальная библиотека Python с большой гибкостью, обеспечивающая надежную производительность и стабильность. Конечно, все это описывает, почему нам стоит использовать NumPy и даже не думать о создании собственного класса. Однако бесплатных завтраков не бывает: как и любой другой инструмент разработчика, у NumPy есть цена. Если стоимость использования NumPy для вашей организации слишком высока, можно рассмотреть другие библиотеки или просто создать собственный класс. Вот некоторые нюансы, которые следует учитывать при работе с NumPy.

- » NumPy используют многие люди, а не только ваша организация. Поэтому критические изменения в пакете NumPy, которые приносят больше пользы большинству, чем вреда меньшинству, всегда будут происходить (согласно <https://numpy.org/neps/nep-0023-backwards-compatibility.html>). Другими словами, нечто столь простое, как обновление NumPy, может привести к сбою вашего приложения.
- » Работа с NumPy — непростое дело. Руководство пользователя занимает 486 страниц (<https://numpy.org/doc/1.20/numpy-user.pdf>) и даже близко не охватывает все аспекты использования NumPy. Как следствие, разработчикам приходится осваивать множество сложных вещей, которые им, возможно, даже не понадобятся.

- » Внесение необходимых изменений в NumPy может оказаться сложным, если учесть рекомендации на <https://numpy.org/doc/stable/dev>. Иными словами, чтобы адаптировать NumPy под нужды вашей организации, вам, возможно, придётся изрядно повозиться – гораздо больше, чем при создании собственного класса.
- » Для понимания деталей реализации NumPy требуется знание C/C++ в дополнение к Python. Например, такие структуры NumPy, как матрицы, основаны на массивах C++, а не на двусвязных списках Python. Преимущество такого подхода в том, что NumPy обеспечивает значительный прирост скорости, но ценой использования нескольких языков программирования. Использование нативного подхода к работе с матрицами медленнее, но проще для понимания.
- » Работа с NumPy отличается от работы с Python, что может создавать проблемы, даже если компромисс кажется проще. Например, если у вас есть две матрицы-строки и вы хотите их сложить, NumPy позволяет сделать это с помощью простого сложения: $A + B$. Тот же код в Python выполняет конкатенацию, то есть две матрицы-строки объединяются. Чтобы сложить две матрицы в Python, можно использовать списковое включение, например: `[a + b for a, b in zip(A, B)]`. Да, подход Python сложнее, но это чистый Python.
- » NumPy использует значения, которые Python не понимает. Самая распространенная жалоба связана с использованием значения NaN (Not-a-Number) для представления отсутствующих значений. Отсутствие поддержки NaN в Python затрудняет сравнение значений. И теперь придется искать решение NumPy для задачи, которая по сути должна решаться на Python. (подробнее см. на <https://medium.com/nerd-for-tech/a-z-with-numpy-library-6269de9c5413>).
- » Использование непрерывного диапазона памяти для хранения данных в NumPy может означать, что NumPy фактически окажется медленнее, если код Python выполняет много вставок и удалений, поскольку данные потребуется сдвигать. То самое свойство, которое делает NumPy таким быстрым, при определенных обстоятельствах может оказаться его погибелью (дополнительную информацию см. на <https://towardsdatascience.com/python-lists-are-sometimes-much-faster-than-numpy-heres-a-proof-4b3dad4653ad>).



СОВЕТ

Существуют и другие аспекты, которые следует учитывать при выборе между стандартным Python и NumPy, но этот список помогает понять, на что стоит обратить внимание. Однако было бы неправильно обобщать и говорить, что всегда нужно использовать Python или всегда создавать собственный класс матрицы. При принятии решений о разработке нужно учитывать приложение в целом и определять, какие компромиссы имеют смысл в конкретной ситуации.

§ 2. Почему стоит использовать классы

Есть много причин для использования классов в Python. Многие из них связаны с методами объектно ориентированного программирования (ООП), такими как:

- инкапсуляция,
- наследование,
- полиморфизм.

Об этих причинах можно прочитать в статьях на <https://www.programiz.com/python-programming/object-oriented-programming> и <https://realpython.com/python3-object-oriented-programming>. Однако эти причины не совсем объясняют, почему вам может понадобиться создавать классы для вашей матричной библиотеки, особенно если вы используете методы функционального программирования (см. пример на <https://www.geeksforgeeks.org/functional-programming-in-python>). Необходимость в классах выходит за рамки использования определенной парадигмы программирования или обычного оправдания «это кажется хорошей идеей», которое люди используют, не совсем понимая, что они имеют в виду под этой фразой. Вот несколько причин использовать класс (или классы) для хранения ваших матричных данных:

- » **Организация.** Матричные данные сами по себе могут стать довольно сложными, и вам не нужно помнить, где вы их разместили. Использование классов помогает организовать данные независимо от того, какую парадигму вы используете для их создания.
- » **Простота доступа.** Размещение ваших матричных данных в одном файле означает, что вы можете получить доступ ко всем из них с помощью единственного импорта.

Однако, если этот импорт огромен, вы теперь выделили значительные ресурсы для потенциального использования всего одной или двух функций. Используя классы для хранения матричных данных, вы можете разделить ваш пакет на управляемые части и импортировать только то, что вам нужно.

- » **Упрощение разработки.** Создание собственных матричных данных требует продвинутых навыков программирования. Сложность задачи может легко оказаться непосильной. Используя подход «разделяй и властвуй», вы можете разбить решаемые проблемы на более мелкие части, что упрощает разработку всего пакета, поскольку вам нужно иметь дело только с одной частью за раз.
- » **Устранение коллизий.** Использование класса объединяет все под одной крышей. Что еще важнее, вы обращаетесь к методам внутри класса, используя одно и то же имя класса, поэтому вероятность возникновения проблем с конфликтами имен при работе с другими пакетами существенно снижается.
- » **Связность.** Объединение всего кода в единый класс способствует созданию связного (единого) подхода к программированию, чтобы вы не использовали один подход для одной части пакета и другой подход – для остальных частей пакета.

§ 3. Создание базового класса

Класс (или группа классов) для работы с матрицами со временем может стать довольно сложным, если вы захотите создать полностью функциональный и универсальный пакет. Насколько сложным он будет, зависит от того, как вы собираетесь его использовать. NumPy – действительно сложная библиотека, потому что она должна универсально удовлетворять потребности большого числа пользователей. Ваш же класс может быть проще, ведь вы точно знаете свои конкретные задачи.



ВНИМАНИЕ

Класс `Matrix` написан с учетом простоты, чтобы вам было легче видеть выполняемые операции. Следовательно, он не включает код проверки типов, проверки размера матрицы или какой-либо обработки ошибок.

Класс, используемый в реальном мире, содержал бы все эти элементы (и даже больше) для обеспечения работы класса с минимальным количеством ошибок. Помните об отсутствии этих функций при работе с классом `Matrix`, проверяя правильность входных данных.

Создание матрицы

Пример класса `Matrix` начинается просто с предоставления кода для создания двумерной матрицы. Вы будете наращивать этот класс по мере продвижения по главе, но важно начать с хорошего фундамента. Следующий код поможет вам создавать двумерные матрицы (наиболее часто используемые), которые не повторяют способ работы NumPy. (Фактически класс `Matrix` показывает альтернативные методы работы с NumPy на протяжении всей главы.)

```
class Matrix:
    rows = 0
    columns = 0
    matrix = []
    matrixRow = []
    dataCount = 0
    matrixList = []
    tempProduct = 0

    def __init__(self, Rows, Columns, Data = []):
        if Data == []:
            Data = [None] * (Rows * Columns)
        self.matrix = []
        self.rows = Rows
        self.columns = Columns
        for i in range(Rows):
            self.matrixRow = []
            for j in range(Columns):
                self.matrixRow.append(
                    Data[self.dataCount])
                self.dataCount += 1
            self.matrix.append(self.matrixRow)
```

Пример делает доступным для просмотра и использования в поддерживающем коде множество переменных, используемых при построении, обслуживании матриц и манипулировании ими. Эти переменные размещены в начале определения

класса, чтобы их было легко найти. Функция `__init__()` (конструктор класса), используемая для создания матрицы, задействует не все эти переменные, но вы увидите их применение в последующих функциях.



ЗАПОМНИТЕ

Конструктор **Matrix** позволяет создавать как пустые матрицы заданного размера, так и матрицы с уже инициализированными значениями, используя список Python. При создании пустой матрицы код заполняет значения **None**, чтобы легко было проверить, инициализирована ли матрица.

Для создания матрицы код определяет отдельные строки, а затем добавляет их в результирующую матрицу. Каждый столбец внутри строки содержит одно значение из списка **Data**. Переменная **dataCount** отслеживает текущую позицию в списке **Data**.

Вывод результирующей матрицы

Данный блок показывает, как работает класс **Matrix**, создавая и выводя матрицу. Следующий код создает и выводит неинициализированную матрицу размером 2×3 :

```
myMatrix = Matrix(2, 3)
print(myMatrix.rows)
print(myMatrix.columns)
print(myMatrix.matrix)
```

Вывод показывает, что свойство **rows** содержит 2, а свойство **columns** содержит 3, как и ожидалось. Результирующая матрица содержит значение **None** для каждого элемента в двумерном выводе:

```
2
3
[[None, None, None], [None, None, None]]
```

Следующий код создает инициализированную матрицу 2×3 , содержащую значения от 0 до 5. Использование функции **range()** упрощает генерацию значений для тестирования:

```
z = list(range(6))
print(z)
myMatrix2 = Matrix(2, 3, z)
print(myMatrix2.matrix)
```

Последующий вывод показывает, как работает функция `__init__()`. Функция, ориентированная на строки, берет каждый элемент входного списка по очереди и создает из него строку матрицы. Вместо этого вы могли бы создать класс, ориентированный на столбцы, в зависимости от ваших конкретных потребностей. Чтобы создать такую же матрицу, используя NumPy, вам нужно импортировать NumPy, а затем задать матрицу с помощью кода вроде этого: `myMatrix = np.array([[1, 2, 3], [4, 5, 6]])`, где форма уже определена.

```
[0, 1, 2, 3, 4, 5]
[[0, 1, 2], [3, 4, 5]]
```



СОВЕТ

Хотя NumPy не предоставляет прямого метода задания формы матрицы во время инициализации, он предоставляет функцию `reshape()` для изменения формы матрицы после инициализации. Класс `Matrix` обеспечивает работу с матрицами фиксированного размера. Нет единственного лучшего способа выполнения задач, есть только способ, который лучше всего работает в конкретной ситуации, поэтому предполагать, что метод, используемый определенной библиотекой Python, всегда лучший — обычно плохая идея.

Доступ к конкретным элементам матрицы

Важно обеспечить метод доступа к конкретным элементам матрицы в вашем классе. Лучший способ решить эту задачу — создать функцию `__getitem__()`, подобную той, что показана здесь:

```
def __getitem__(self, index):
    return self.matrix[index]
```



ЗАПОМНИТЕ

Функция `__getitem__()` — это лишь одна из длинного списка специальных методов, доступных для использования в вашем классе. (Убедитесь, что вы добавляете `getitem()` и функции, которые будут рассмотрены в следующих блоках, непосредственно в класс `Matrix`, а не на верхний уровень.) Вы можете прочитать о других специальных методах на странице <https://docs.python.org/3/reference/datamodel.html#special-method-names>. Эти специальные методы предоставляют специфичный для Python способ выполнения таких задач, как перегрузка операторов, поэтому их следует использовать при создании класса всякий раз, когда это возможно.

В случае с `__getitem__()` (https://docs.python.org/3/reference/datamodel.html#object.__getitem__) код обеспечивает метод получения среза матрицы. Все, что вам нужно сделать, — это вернуть часть матрицы, на которую ссылается `index`. Он может быть любым допустимым срезом Python, как показано в следующем коде:

```
print(myMatrix2[1])
print(myMatrix2[1][2])
```

Вывод демонстрирует, что Python интерпретирует индекс соответствующим образом в зависимости от того, что пользователь предоставляет в качестве входных данных (первый индекс — это строка, а второй задает и строку, и столбец):

```
[3, 4, 5]
5
```

Выполнение скалярного сложения и сложения матриц

Помимо базовой необходимости создавать, печатать и делать срезы матрицы, важно также выполнять некоторые основные манипуляции, такие как сложение. Неудивительно, что для выполнения этой задачи используется специальный метод `__add__()`, как показано здесь:

```
def __add__(self, Value):
    self.matrixList = []
    if type(Value) == list:
        for i in range(self.rows):
            for j in range(self.columns):
                self.matrixList.append(
                    self.matrix[i][j] + Value[i][j])
    else:
        for i in range(self.rows):
            for j in range(self.columns):
                self.matrixList.append(
                    self.matrix[i][j] + Value)
    return Matrix(self.rows, self.columns,
                  self.matrixList)
```

Код охватывает два случая. В первом случае `Value` содержит матрицу, поэтому код должен сложить одну матрицу с другой матрицей. Во втором случае `Value` содержит целое число, которое добавляется к каждому

элементу матрицы. Независимо от того, какую форму входных данных предоставляет пользователь, значения складываются с использованием `matrixList`, который является внутренней переменной списка.

В какой-то момент `matrixList` содержит список правильных значений. Затем код создает новую матрицу `Matrix` правильного размера и возвращает ее вызывающей программе. Такой подход гарантирует, что у возвращаемого объекта — тип `Matrix`, а не тип `list`, что произошло бы, если бы функция не выполняла этот последний шаг.

Функция `__add__()` работает как для скалярного сложения, так и для сложения матриц, что продемонстрировано в следующем коде:

```
myMatrix2 += 2
print(myMatrix2.matrix)
myMatrix2 += [[2, 4, 6], [8, 10, 12]]
print(myMatrix2.matrix)
```

Результат показывает правильные значения сложения. (Вы можете проверить результаты с помощью NumPy, которая использует тот же метод сложения, что и класс `Matrix`.)

```
[[2, 3, 4], [5, 6, 7]]
[[4, 7, 10], [13, 16, 19]]
```

Выполнение умножения

Любой практичный класс для работы с матрицами должен поддерживать два вида умножения. Первый — это поэлементное произведение, которое просто выполняет умножение поэлементно. Второй — это скалярное произведение, которое часто используется в линейной алгебре. В следующих блоках показано, как выполнять оба типа умножения с использованием класса `Matrix`.

Поэлементное произведение

Поэлементное произведение используется, когда нужно объединить две матрицы путем умножения, а не с помощью какой-либо другой операции, например сложения. К примеру, у вас может быть одна матрица, показывающая количество отработанных часов за каждый день, и другая матрица, показывающая почасовую оплату за каждый рабочий день. Использование поэлементного произведения создаст третью матрицу, содержащую общую заработную плату за каждый рабочий день.

Для получения такого результата используется функция `__mul__()`, показанная здесь:

```
def __mul__(self, MatrixIn):
    self.matrixList = []
    for i in range(self.rows):
        for j in range(self.columns):
            self.matrixList.append(
                self.matrix[i][j] * MatrixIn[i][j])
    return Matrix(self.rows, self.columns,
                  self.matrixList)
```

Код просто умножает элементы одной матрицы на соответствующие элементы другой матрицы. Для этого в классе `Matrix` обе матрицы должны быть одинакового размера. При работе с NumPy у вас есть возможность выполнять скалярное или векторное умножение. Очевидно, вы могли бы добавить дополнительный код, чтобы сделать это возможным и в классе `Matrix`. Вот пример использования класса `Matrix`:

```
A = Matrix(2, 3, [1, 2, 3, 4, 5, 6])
B = Matrix(2, 3, [1, 2, 3, 4, 5, 6])
print(A.matrix)
print(B.matrix)
print((A * B).matrix)
```



ЗАПОМНИТЕ

Обратите внимание, что для получения внутреннего произведения в классе `Matrix` используется оператор. То же самое верно и для NumPy: для получения внутреннего произведения используется `*` ($\times$). Вот результат умножения:

```
[[1, 2, 3], [4, 5, 6]]
[[1, 2, 3], [4, 5, 6]]
[[1, 4, 9], [16, 25, 36]]
```

Как показано, в данном случае умножение выполняется просто. Например, `A[0][1] * B[0][1]` равняется 2×2 , что отображается как 4 в результате.

Скалярное произведение

Многим людям трудно понять, почему скалярное произведение полезно, не говоря уже о том, как его реализовать. Скалярное произведение оказывается полезным при выполнении таких задач, как вычисление общего объема продаж. Начнем с вектора `Price`, который содержит цены на три вида фруктов:

Яблоки	Вишня	Груши
1	2	1

Матрица **Sales** содержит количество фунтов каждого товара, проданного за день:

	Понедельник	Вторник	Среда	Четверг	Пятница
Яблоки	5	3	4	3	2
Вишня	2	3	3	4	4
Груши	1	2	4	2	3



ЗАПОМНИТЕ

Различное расположение вектора относительно матрицы важно при вычислении скалярного произведения. Чтобы получить общий объем продаж, нужно умножить вектор на столбцы таблицы. Например, чтобы найти итог за понедельник, выполняется следующее вычисление: $(1 \times 5) + (2 \times 2) + (1 \times 1)$, что равно 10 \$. Вывод для всех дней будет выглядеть примерно так.

[[10, 11, 14, 13, 13]]

Работа с матрицей немного сложнее. Допустим, у вас есть матрица **Prices**, которая содержит цены на три вида фруктов для каждого дня недели:

	Яблоки	Вишня	Груши
Понедельник	1	2	1
Вторник	1	3	2
Среда	2	2	1
Четверг	2	3	2
Пятница	1	2	2

Итак, в понедельник яблоки продавались по 1 \$ за фунт, вишня по 2 \$ за фунт и груши по 1 \$ за фунт. Сумма продаж меняется по дням недели, поэтому нельзя сказать, что стоимость каждого продукта постоянна. Результат такого расчета выглядел бы примерно так:

```
[[10, 11, 14, 13, 13],
 [13, 16, 21, 19, 20],
 [15, 14, 18, 16, 15],
 [18, 19, 25, 22, 22],
 [11, 13, 18, 15, 16]]
```

Выходные значения теперь появляются по диагонали, поэтому второе выходное значение будет равно 16, или $(1 \times 3) + (3 \times 3) + (2 \times 2)$, что соответствует значениям вторника в каждой матрице. Итог среды — 18, четверга — 22, а пятницы — 16. Именно так работает NumPy. Ваш пользовательский класс может просто выводить диагональные значения, чтобы никому не пришлось ничего интерпретировать.

Код для выполнения этих вычислений должен быть обобщенным, чтобы принимать в качестве входных данных как вектор, так и матрицу, поэтому он выглядит так:

```
def dot(self, MatrixIn):
    self.matrixList = []
    for i in range(self.rows):
        for j in range(MatrixIn.columns):
            tempProduct = 0
            for k in range(self.columns):
                tempProduct += self.matrix[i][k] * \
                    MatrixIn[k][j]
            self.matrixList.append(tempProduct)
    return Matrix(self.rows, MatrixIn.columns,
                  self.matrixList)
```

Функциональность срезов, предоставляемая Python, автоматически обрабатывает различия между векторным и матричным умножением. Однако количество столбцов первой сущности (вектора или матрицы) всегда должно совпадать с количеством строк второй сущности. Тестовый код для этого примера выглядит так для вектора:

```
Price = Matrix(1, 3, [1, 2, 1])
Sales = Matrix(3, 5,
               [5, 3, 4, 3, 2, 2, 3, 3, 4, 4, 1, 2, 4, 2, 3])
print(Price.matrix)
print(Sales.matrix)
print(Price.dot(Sales).matrix)
```

А для матрицы тестовый код выглядит так:

```
Prices = Matrix(5, 3,
                [1, 2, 1, 1, 3, 2, 2, 2, 1, 2, 3, 2, 1, 2, 2])
```

```
print(Prices.matrix)
print(Prices.dot(Sales).matrix)
```

В обоих случаях результат будет точно таким же, как при использовании NumPy.

§ 4. Манипулирование матрицей

Есть много способов манипулирования матрицами в зависимости от желаемого результата. Фактически некоторые из этих методов встречаются на протяжении всей книги. Однако одни манипуляции используются чаще других. В следующих блоках рассматриваются три такие операции: транспонирование, вычисление определителя и уплощение (преобразование в одномерный список).

Транспонирование матрицы

В блоке «Скалярное произведение» ранее в главе рассматривалось, как получить скалярное произведение двух матриц. В том блоке мы изучали, как подсчитать общие продажи фруктов, но трудно переоценить области применения скалярного произведения. Чтобы получить скалярное произведение, матрицы должны быть ориентированы определенным образом. Однако может оказаться, что ваша матрица ориентирована неправильно, и именно здесь в игру вступает транспонирование. В этом блоке рассматривается простое транспонирование, при котором строки становятся столбцами. Для выполнения этой задачи используется следующий код:

```
def transpose(self):
    self.matrixList = []
    for i in range(self.columns):
        for j in range(self.rows):
            self.matrixList.append(self.matrix[j][i])
    return Matrix(self.columns, self.rows,
                  self.matrixList)
```

По сути, процесс включает копирование матрицы, но так, что строки и столбцы меняются местами. Теперь `matrixList` содержит значения в порядке следования столбцов, а не строк (порядок строк является нормальным). Чтобы создать правильный вывод, вызов `Matrix()` также

должен поменять местами столбцы и строки, чтобы форма результирующей матрицы отражала новый порядок. Тестовый код для этого примера выглядит так:

```
print(A.matrix)
print(A.transpose().matrix)
```

Вывод отражает изменение ориентации:

```
[[1, 2, 3], [4, 5, 6]]
[[1, 4], [2, 5], [3, 6]]
```

Вычисление определителя

В следующих блоках обсуждается, как создать код для вычисления определителя (который применим только к квадратным матрицам). Конечно, вам может понадобиться несколько пояснений о том, что такое определитель и зачем его нужно вычислять, поэтому следующий блок также охватывает эти детали.

Почему матрица так определена?

Определитель матрицы — это особое число, которое используется во множестве приложений, например при вычислении обратной матрицы (<https://www.mathsisfun.com/algebra/matrix-inverse-minors-cofactors-adjugate.html>). Обратная матрица — это набор значений, которые при умножении на исходную матрицу дают единичную матрицу. Да, звучит довольно запутанно, поэтому вот упрощенная версия: если у вас есть значение 10 и вы вычисляете его обратное значение, которое равно $1/10$, то умножение $10 \times 1/10$ дает результат 1. Единичная матрица подобна значению 1 для матриц. Если вам действительно интересно узнать больше об обратных матрицах и единичной матрице, вы можете найти отличное объяснение по адресу <https://www.mathsisfun.com/algebra/matrix-inverse.html>. Иногда для выполнения определенных задач нужно найти обратную матрицу, что требует вычисления определителя.



ПРИМЕЧАНИЕ

Есть много способов найти определитель матрицы. В этом примере используется разложение Лапласа, первоначально разработанное Пьером-Симоном Лапласом (<https://www.britannica.com/biography/Pierre-Simon-marquis-de-Laplace>), поскольку этот подход очень удобно реализуется с помощью рекурсии. Это не самый быстрый способ

выполнить вычисление, но он наиболее понятен. Если хотите ознакомиться с одной из альтернатив, посмотрите упрощенный метод, описанный на <https://www.studypug.com/algebra-help/the-determinant-of-a-3-x-3-matrix-general-and-shortcut-method>.



СОВЕТ

В таких ситуациях полезно иметь под рукой специальный калькулятор для проверки результатов работы вашего кода. Сайт Matrix Reshish (<https://matrix.resish.com/determinant.php>) предоставляет калькуляторы для вычисления определителя матрицы, а еще много других полезных инструментов, таких как транспонирование матрицы и определение ранга. Калькулятор Symbolab (<https://www.symbolab.com/solver/matrix-determinant-calculator>) особенно полезен тем, что показывает пошаговое решение задачи нахождения определителя, что может помочь при написании кода. Он также демонстрирует много других методов работы с матрицами и решения проблем, с которыми вы можете столкнуться.

Создание необходимого вспомогательного кода

Код для этого примера начинается с функции `copyMatrix()`, которая позволяет копировать часть матрицы в новую матрицу, используя методы срезов. Необходимость копирования частей матрицы обусловлена тем, что разложение Лапласа сводит задачу вычисления определителя к простому случаю — матрице 2×2 . Следовательно, код требует метод получения необходимой матрицы 2×2 , что и является целью функции `copyMatrix()`:

```
def copyMatrix(self):
    for i in range(self.rows):
        for j in range(self.columns):
            self.matrixList.append(self.matrix[i][j])
    return Matrix(self.rows, self.columns,
                  self.matrixList)
```



ЗАПОМНИТЕ

Как показано, код просто создает новую матрицу, которая в точности совпадает с входной матрицей. Причина, по которой нужно использовать именно такой подход, заключается в том, что копирование матрицы любым другим способом может привести к тому, что две переменные будут указывать на одну и ту же область памяти. Изменение одной переменной неизбежно приведет к изменению содержимого другой переменной. Функция `copyMatrix()` гарантирует, что новая матрица действительно указывает на новую область памяти.

Выполнение вычисления

Чтобы вычислить определитель матрицы 2×2 , нужно перемножить элементы по диагоналям, а затем вычесть из первого произведения второе. Например, начнем с матрицы, которая выглядит так:

```
[[1, 2],  
 [3, 4]]
```

В этом случае определитель равен $(1 \times 4) - (3 \times 2)$, или $4 - 6$, что дает результат -2 . Имея в виду рекурсию, рассмотрим теперь вычисление для следующей матрицы, 3×3 :

```
[[2, 5, 1],  
 [5, 6, 7],  
 [10, 9, 8]]
```

Здесь мы фактически имеем дело с тремя матрицами 2×2 , которые можно разложить следующим образом:

- 1** $2 \times ((6 \times 8) - (9 \times 7)) = 2 \times (48 - 63) = -30;$
- 2** $5 \times ((5 \times 8) - (10 \times 7)) = 5 \times (40 - 70) = -150;$
- 3** $1 \times ((5 \times 9) - (10 \times 6)) = 1 \times (45 - 60) = -15.$

Чтобы получить окончательный результат, выполняем математические действия так:

```
-30 - -150 + -15 = 105
```

Последняя часть может показаться немного сложной, поскольку не сразу понятно, нужно складывать или вычитать результаты подматриц 2×2 . На самом деле, просто чередуем сложение и вычитание, всегда начиная с вычитания, как показано выше. Самое важное, что нужно усвоить в данном случае, — мы всегда умножаем верхнюю строку на подматрицу 2×2 . Работа с матрицей 4×4 — это просто расширение этого принципа. Для выполнения этой задачи класс содержит следующий код:

```
def determinant(self, Result=0):  
    # Address the simplest case first, the 2 X 2 matrix.  
    if len(self.matrix) == 2:  
        twoOut = self.matrix[0][0] * self.matrix[1][1] - \  
            self.matrix[1][0] * self.matrix[0][1]  
        return twoOut  
  
    # Determine the number of rows in a matrix larger  
    # than 2 X 2.
```



```

rows = list(range(len(self.matrix)))

# Process each focus column in turn.
for focus in rows:

    # Create a copy of the matrix.
    submatrix = self.copyMatrix()

    # Remove the first row of the submatrix.
    submatrix.matrix = submatrix.matrix[1:]

    # Obtain the number of remaining rows to
    # process.
    subrows = len(submatrix.matrix)

    # Create the next smaller size matrix by slicing
    # out the focus rows.
    for i in range(subrows):
        submatrix.matrix[i] = \
            submatrix.matrix[i][0:focus] + \
            submatrix.matrix[i][focus+1:]

    # Determine the sign to use when performing the
    # multiplication.
    sign = (-1) ** (focus % 2)

    # Call the determinant() function recursively
    # with each smaller matrix.
    subdeterminant = submatrix.determinant()

    # Total the returns from the recursive calls.
    Result += sign * self.matrix[0][focus] * \
        subdeterminant

return Result

```

Код состоит из двух частей. Первая часть — это простой случай матрицы 2×2 , который следует описанному ранее процессу.

Вторая часть начинается с создания копии текущей матрицы в `submatrix`. Затем код получает только верхнюю строку в `submatrix`. Далее он вырезает матрицу следующего меньшего размера. Так, если вы работаете с матрицей 3×3 , код создает три матрицы 2×2 , как было описано ранее. Затем код определяет, нужно прибавлять или вычитать каждое из вычислений подматрицы, и рекурсивно вызывает `determinant()`. Последний шаг — сложение или вычитание результатов каждого рекурсивного вызова.

Уплотнение матрицы (преобразование в одномерный список)

Многие алгоритмы требуют преобразования матрицы в одномерный список для получения желаемого результата. Вы можете захотеть «сплющить» матрицу, чтобы нагляднее увидеть результат работы алгоритма или использовать его определенным образом. Что интересно, эта конкретная задача часто встречается на собеседованиях. Независимо от причины, по которой вам нужно преобразовать матрицу в одномерный список, есть много способов выполнить эту задачу, и вы можете найти их обсуждение в интернете. Вот метод, используемый в классе `Matrix`:

```
def flatten(self):
    self.matrixList = []
    for i in range(self.rows):
        for j in range(self.columns):
            self.matrixList.append(self.matrix[i][j])
    nestedResult = Matrix(1, self.rows * self.columns,
                          self.matrixList)
    nestedResult.matrix = nestedResult.matrix[0]
    return nestedResult
```

Как и другие элементы класса, этот метод не гарантирует самого быстрого результата, но предоставляет понятный способ решения задачи. Код просто создает `matrixList`, который последовательно добавляет каждый элемент матрицы. Однако результатом этого вложенного цикла является список, а не `Matrix`, поэтому следующий шаг — создание `Matrix`, содержащей одномерный список.

Проблема с выводом на этом этапе заключается в том, что полученная `Matrix` фактически содержит список, вложенный в другой список. Чтобы исправить результат, код присваивает `nestedResult.matrix` значение `nestedResult.matrix[0]`. Теперь результат представляет собой действительно «сплюсненную» матрицу типа `Matrix`. Вот тестовый код для этого примера:

```
A = Matrix(3, 3, [1, 2, 3, 4, 5, 6, 7, 8, 9])
print(A.matrix)
print(A.flatten().matrix)
```

Вывод показывает, что исходная матрица действительно преобразована в одномерный список в выходных данных:

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```



ПОНИМАНИЕ НЕОБХОДИМОСТИ СОРТИРОВКИ И ПОИСКА

В ЭТОМ РАЗДЕЛЕ...

Складывание и накопление данных

Взаимодействие с деревьями и графами

Использование сортировки слиянием
и быстрой сортировки

Использование деревьев поиска и кучи

Хеширование данных

В ЭТОЙ ГЛАВЕ...

- » Определение, почему данным нужна структура
- » Работа со стеками, очередями, списками и словарями
- » Использование деревьев для организации данных
- » Использование графов для представления данных с отношениями

ГЛАВА 6

Структурирование данных

Необработанные данные — это просто необработанные данные. Они никак не структурированы и не очищены. Прежде чем сможете что-либо сделать с большинством данных, вы должны как-то структурировать их, чтобы начать видеть, что они содержат (а иногда и чего в них нет). Первый параграф главы посвящен необходимости преобразования необработанных данных в структурированные.

Python предоставляет доступ к ряду организационных структур для данных. В книге эти структуры, особенно стеки, очереди и словари, используются во многих примерах. Каждая структура данных предоставляет различные средства работы с данными и различные наборы инструментов для выполнения таких задач, как сортировка данных в определенном порядке. В главе представлены наиболее распространенные методы организации данных, включая деревья и графы (которые настолько важны, что им посвящены отдельные блоки).



ЗАПОМНИТЕ

Вам не нужно вручную набирать исходный код для этой главы. На самом деле, использование загружаемого исходного кода намного проще. Исходный код главы можно найти в файлах `A4D2E; 06; Graphs.ipynb`, `A4D2E; 06; Remediation.ipynb`, `A4D2E; 06; Stacks, Queues, and Dictionaries.ipynb` и `A4D2E; 06; Trees.ipynb` из загружаемых материалов. Подробнее о том, как найти эти файлы, смотрите во введении.

§ 1. Определение необходимости структуры

Структурирование данных подразумевает их организацию так, чтобы все они имели одинаковые атрибуты, представление и компоненты. Например, вы можете получить данные из одного источника, где даты представлены в виде строк, а из другого источника — в виде объектов даты. Чтобы использовать эту информацию, нужно привести типы данных к единому виду. Источники данных также могут по-разному структурировать информацию. В одном источнике фамилия и имя могут находиться в едином поле, а в другом — в отдельных полях. Важная часть структурирования данных — их организация. При этом вы не меняете сами данные — просто делаете их удобнее для использования. (Структурирование данных отличается от их очистки или преобразования, при которых значения иногда действительно изменяются для конвертации одного типа данных в другой, или происходит потеря точности, например с датами, при переносе между источниками данных.)

Структура — важнейший элемент в работе алгоритмов. Как показано в примере бинарного поиска в главе 4, реализовать алгоритм с использованием структурированных данных намного проще, чем пытаться разобраться, как интерпретировать данные в коде. Например, алгоритм бинарного поиска требует, чтобы данные были отсортированы. Попытка выполнить необходимые сравнения с неотсортированными данными потребовала бы гораздо больше усилий и могла бы даже оказаться невыполнимой. Учитывая это, вам нужно продумать структурные требования к данным, которые вы используете в своих алгоритмах, как описано в следующих блоках.

Облегчаем понимание содержимого

Для работы с данными крайне важно понимать их содержимое. Алгоритм поиска работает только тогда, когда вы понимаете набор данных и знаете, что нужно искать с помощью алгоритма. Поиск слов в наборе данных, содержащем числа, — это невыполнимая задача, которая всегда приводит к ошибкам. Тем не менее ошибки поиска, возникающие из-за непонимания содержимого набора данных, случаются часто, даже с лучшими поисковыми системами. Люди делают предположения о содержимом наборов данных, которые приводят к сбоям алгоритмов. Следовательно, чем лучше вы можете видеть и понимать содержимое благодаря структурированному форматированию, тем проще становится успешно выполнять алгоритмические задачи.

Однако даже в анализе содержимого часто возникают ошибки при взаимодействии людей и компьютеров. Например, если вы попытаетесь найти число, представленное в виде строки, в наборе данных, где числа хранятся как целые значения, поиск не даст результатов. Компьютеры не выполняют автоматическое преобразование между строками и целыми числами, как это делают люди. На самом деле, компьютеры воспринимают все как числа, а строки — это лишь интерпретация этих чисел, заданная программистом. Поэтому при поиске строки «1» компьютер воспринимает это как запрос числа 49 при использовании ASCII-кодировки. Чтобы найти числовое значение 1, нужно искать именно целое число 1.



ЗАПОМНИТЕ

Структура позволяет также выявлять нюансы в данных. Например, номер телефона может быть записан в формате (555)555-1212. Если выполнить поиск или другую алгоритмическую операцию, используя формат 555-555-1212, поиск может не дать результатов из-за различий в записи кода региона в начале — используется 555- вместо (555). Такие проблемы вызывают серьезные затруднения, поскольку большинство людей воспринимают эти две формы записи как идентичные, но компьютер — нет. Попытки навязать людям определенный формат редко работают и обычно приводят к разочарованию, что делает использование алгоритма еще более сложным. Поэтому структурирование данных через их преобразование становится еще более важным.

Сопоставление данных из разных источников

Взаимодействие с данными из одного источника — это одна проблема, а работа с данными из нескольких источников — совсем другая. Однако сегодня наборы данных обычно поступают из разных источников, поэтому нужно понимать сложности, которые могут возникнуть при использовании множественных источников данных. При работе с несколькими источниками данных необходимо:

- » **определить, содержат ли оба набора данных всю необходимую информацию.** Маловероятно, что два разработчика создадут наборы данных, содержащие в точности одинаковые данные, в одном и том же формате, одного типа и в том же порядке. Следовательно, нужно проверить, предоставляют ли наборы данных нужную информацию, или требуется выполнить очистку данных для получения желаемого результата, как описано в следующем блоке;
- » **проверить оба набора данных на предмет проблем с типами данных.** В одном наборе даты могут быть введены

как строки, а в другом – как настоящие объекты дат. Несоответствия между типами данных вызовут проблемы для алгоритма, который ожидает данные в одной форме, а получает их в другой;

- » **убедиться, что все наборы данных одинаково интерпретируют элементы данных.** Данные, созданные одним источником, могут иметь иное значение, чем данные из другого источника. Например, размер целого числа может различаться в разных источниках: вы можете встретить 16-битное целое число из одного источника и 32-битное – из другого. Меньшие значения имеют одинаковый смысл, но 32-битное целое число может содержать бо льшие значения, что может вызвать проблемы в работе алгоритма;
- » **проверить атрибуты данных.** У элементов данных есть определенные атрибуты (подробнее см. на https://www.w3schools.com/python/python_datatypes.asp). В главе 4 отмечается, что эта интерпретация может меняться при использовании `numpy`. На самом деле, атрибуты данных меняются между средами, а разработчики могут изменять их еще больше, создавая пользовательские типы данных. Чтобы объединить данные из разных источников, вы должны понимать эти атрибуты, чтобы правильно интерпретировать данные.



ЗАПОМНИТЕ

Чем больше времени вы потратите на проверку совместимости данных из каждого источника, которые вы хотите использовать для набора данных, тем меньше вероятность столкнуться с проблемами при работе с алгоритмом. Проблемы несовместимости данных не всегда проявляются как явные ошибки. В некоторых случаях несовместимость может вызвать другие проблемы, например ошибочные результаты, которые выглядят правильно, но предоставляют вводящую в заблуждение информацию.

Объединение данных из нескольких источников не всегда означает создание нового набора данных, который выглядит в точности как исходные наборы. В некоторых случаях вы создаете агрегаты данных или выполняете другие формы манипуляций для создания новых данных из существующих. Анализ принимает самые разные формы, и некоторые из более экзотических форм могут привести к серьезным ошибкам при неправильном использовании. В качестве крайнего примера представьте, что произойдет при объединении информации о пациентах из нескольких источников и последующем создании объединенных записей пациентов в новом источнике данных со всевозможными несоответствиями. Пациент без истории определенного заболевания может оказаться с записями, показывающими диагностику и лечение этого заболевания.

Учет необходимости исправления данных

После обнаружения проблем с набором данных вам нужно его исправить, чтобы он правильно работал с используемыми алгоритмами. Например, при работе с конфликтующими типами данных вы должны изменить типы данных каждого источника так, чтобы они совпадали, а затем создать единый источник данных, используемый алгоритмом. Большая часть этого исправления, хотя и требует времени, является прямолинейной. Вам просто нужно перед внесением изменений убедиться, что вы понимаете данные, а это означает способность видеть содержимое в контексте того, что вы планируете с ним делать. Однако нужно учитывать два особых случая: дублирование данных и отсутствующие данные. В следующих блоках показано, как справляться с такими проблемами.

Работа с дублированием данных

Дублирование данных происходит по разным причинам. Некоторые из них очевидны. Пользователь может ввести одни и те же данные несколько раз. Отвлекающие факторы заставляют людей терять место в списке, а иногда два пользователя вводят одну и ту же запись. Некоторые источники дублирования менее очевидны. Объединение двух или более наборов данных может создать множественные записи, когда данные появляются в нескольких местах. Дублирование данных также может возникнуть при использовании различных методов преобразования для создания новых данных из существующих источников. К счастью, такие пакеты, как Pandas, позволяют удалять дублирующиеся данные, как показано в следующем примере:

```
import pandas as pd

df = pd.DataFrame({'A': [0,0,0,0,0,1,0],
                   'B': [0,2,3,5,0,2,0],
                   'C': [0,3,4,1,0,2,0]})

print(df, "\n")

df = df.drop_duplicates()
print(df)
```

При запуске этого кода вы увидите следующий результат, который сначала показывает исходные данные, а затем — данные с удаленными дубликатами:


```

A B C
0 0 0 0
1 0 2 3
2 0 3 4
3 0 5 1
4 0 0 0
5 1 2 2
6 0 0 0

```

```

A B C
0 0 0 0
1 0 2 3
2 0 3 4
3 0 5 1
5 1 2 2

```

Функция `drop_duplicates()` удаляет повторяющиеся записи, найденные в строках 4 и 6 в этом примере. Загрузив данные из источника в `DataFrame pandas`, вы можете быстро удалить лишние записи, чтобы дубликаты не искажали результаты работы используемых алгоритмов.

Работа с отсутствующими значениями

Отсутствующие значения также могут исказить результаты работы алгоритма. Фактически они могут вызвать странное поведение некоторых алгоритмов или даже привести к ошибке. Есть много вариантов работы с отсутствующими значениями. Например, можно установить для них стандартное значение, такое как 0 для целых чисел. Конечно, использование стандартной настройки также может исказить результаты. Другой подход — использовать среднее значение всех значений, что обычно делает отсутствующие значения нейтральными. Использование среднего значения показано в следующем примере:

```

import pandas as pd
import numpy as np
df = pd.DataFrame({'A': [0,0,1,None],
                    'B': [1,2,3,4],
                    'C': [np.NaN,3,4,1]},
                  dtype=int)

print(df, "\n")

values = pd.Series(df.mean(), dtype=int)
print(values, "\n")

```

```
df = df.fillna(values)
print(df)
```

При запуске этого кода вы сначала увидите исходные данные, включающие как **NaN**, так и **None**, а затем — данные с заполненными отсутствующими значениями:

```
   A  B  C
0  0  1 NaN
1  0  2  3
2  1  3  4
3 None 4  1
```

```
A    0
B    2
C    2
dtype: int32
```

```
   A  B  C
0  0  1  2
1  0  2  3
2  1  3  4
3  0  4  1
```

Функция `fillna()` позволяет избавиться от отсутствующих значений, независимо от того, являются они нечисловыми (**NaN**) или просто отсутствующими (**None**). Вы можете предоставить значения для отсутствующих данных в нескольких формах. В этом примере используется серия, содержащая среднее значение для каждого отдельного столбца данных (как это делается при работе с базой данных).



ЗАПОМНИТЕ

Обратите внимание, что код тщательно создан, чтобы не вносить ошибки в результаты, гарантируя, что у `values` правильный тип данных. Обычно функция `mean()` выводит значения с плавающей запятой, но вы можете принудительно привести заполняемый ею ряд к нужному типу. В результате выходные данные не только не содержат пропущенных значений, но и представлены в корректном типе данных.

Понимание других проблем очистки данных

Очистка данных может принимать различные формы. Приложения не всегда обеспечивают проверку правил ввода данных, поэтому пользователи могут вводить неправильные названия штатов или регионов. Пользователи также могут допускать орфографические ошибки. Кроме

того, значения иногда выходят за допустимые пределы или просто невозможны в данной ситуации.

Не всегда удастся полностью очистить данные с первой попытки. Часто о проблеме становится известно после запуска алгоритма, когда замечаешь, что результаты искажены или алгоритм вообще не работает (даже если он работал на подмножестве данных). В случае сомнений проверяйте данные на предмет необходимости очистки.

§ 2. Упорядочивание и нагромождение данных

Python предоставляет несколько методологий хранения данных (см. на https://www.w3schools.com/python/python_lists.asp в качестве примера). Как вы уже видели в этой главе, пакеты часто предлагают дополнительные методы хранения. И NumPy, и Pandas предоставляют альтернативные способы хранения, которые стоит рассмотреть при решении различных задач структурирования данных.



ЗАПОМНИТЕ

Распространенная проблема хранения данных заключается не только в необходимости их хранения, но и в том, что их нужно хранить в определенном порядке, чтобы иметь возможность доступа к ним при необходимости. Например, вы захотите, чтобы первый элемент, помещенный в стек элементов для обработки, также был первым обрабатываемым элементом. С учетом этой проблемы упорядочивания данных в следующих блоках описываются стандартные методы Python для обеспечения упорядоченного хранения данных, позволяющие организовать определенный порядок обработки.

Упорядочивание в стеках

Стек обеспечивает хранение данных по принципу «последним пришел — первым ушел» (LIFO). Пакет NumPy предоставляет реальную реализацию стека. Кроме того, Pandas связывает стеки с такими объектами, как `DataFrame`. Однако оба пакета скрывают детали реализации стека, а понимание принципов работы стека действительно полезно. Поэтому в следующем примере реализуется стек с использованием стандартного списка Python:


```

MyStack = []
StackSize = 3
def DisplayStack():
    print("Stack currently contains:")
    for Item in MyStack:
        print(Item)

def Push(Value):
    if len(MyStack) < StackSize:
        MyStack.append(Value)
    else:
        print("Stack is full!")

def Pop():
    if len(MyStack) > 0:
        print("Popping: ", MyStack.pop())
    else:
        print("Stack is empty.")

# Test the stack functionality.

Push(1)
Push(2)
Push(3)
DisplayStack()

Push(4)

Pop()
DisplayStack()

Pop()
Pop()
Pop()

```

Этот код состоит из двух частей. Первая часть предоставляет реализацию стека. Вторая — содержит код для тестирования реализации. При запуске этого кода вы увидите следующий вывод:

```

Stack currently contains:
1
2
3
Stack is full!
Popping: 3
Stack currently contains:

```

```
1
2
Popping: 2
Popping: 1
Stack is empty.
```

Пример гарантирует, что стек поддерживает целостность данных и работает с ними в ожидаемом порядке. Код опирается на простые манипуляции со списками, но эффективно обеспечивает представление стека, которое можно использовать для любых нужд.

Использование очередей

В отличие от стеков, очереди являются структурами данных типа «первым пришел — первым ушел» (FIFO). Как и в случае со стеками, готовые реализации можно найти во многих пакетах, включая NumPy и Pandas. К счастью, в Python также есть специальная реализация очереди, которая демонстрируется в следующем коде:

```
import queue

MyQueue = queue.Queue(3)

print("Queue empty: ", MyQueue.empty())

MyQueue.put(1)
MyQueue.put(2)
MyQueue.put(3)
print("Queue full: ", MyQueue.full())

print("Popping: ", MyQueue.get())
print("Queue full: ", MyQueue.full())

print("Popping: ", MyQueue.get())
print("Popping: ", MyQueue.get())
print("Queue empty: ", MyQueue.empty())
```

При запуске этого примера вы увидите следующий результат:

```
Queue empty: True
Queue full: True
Popping: 1
Queue full: False
Popping: 2
```

```
Popping: 3  
Queue empty: True
```

Использование встроенной очереди требует гораздо меньше кода, чем создание стека с нуля с помощью списка. Но обратите внимание на различия в выводе. В примере со стеком значения 1, 2 и 3 помещаются в стек, поэтому первым извлекается значение 3. Однако в этом примере при добавлении значений 1, 2 и 3 в очередь первым извлекается значение 1.

Поиск данных с помощью словарей

Создание и использование словаря во многом похоже на работу со списком, за исключением того, что теперь нужно определять пару «ключ — значение». Большое преимущество этой структуры данных заключается в том, что словари могут быстро предоставлять доступ к конкретным элементам данных с помощью ключа. Существуют ограничения на типы ключей, которые можно использовать. Вот особые правила создания ключа:

- » **Ключ должен быть уникальным.** При вводе повторяющегося ключа информация из второй записи имеет приоритет; вторая запись заменяет первую.
- » **Ключ должен быть неизменяемым.** Это правило означает, что в качестве ключа можно использовать строки, числа или кортежи. Однако нельзя использовать список в качестве ключа.

Разница между изменяемыми и неизменяемыми значениями в том, что неизменяемые значения не могут меняться. Например, чтобы изменить значение строки, Python фактически создает новую строку с новым значением и дает ей то же имя, что и у старой. Затем старая строка уничтожается.



СОВЕТ



ЗАПОМНИТЕ

Словари Python являются программной реализацией структуры данных, называемой *хеш-таблицей*, — массивом, который сопоставляет ключи со значениями. В главе 7 подробно объясняются хеши и то, как их использование может помочь словарям работать быстрее. На значения, которые вы предоставляете, ограничений нет. Значением может быть любой объект Python, поэтому вы можете использовать словарь для доступа к записи сотрудника или другим сложным данным. Следующий пример поможет вам лучше понять, как использовать словари:


```
Colors = {"Sam": "Blue", "Amy": "Red", "Sarah": "Yellow"}

print(Colors["Sarah"])
print(Colors.keys())

for Item in Colors.keys():
    print("{0} likes the color {1}."
          .format(Item, Colors[Item]))
Colors["Sarah"] = "Purple"
Colors.update({"Harry": "Orange"})
del Colors["Sam"]

print(Colors)
```

При запуске этого кода вы увидите следующий результат:

```
Yellow
dict_keys(['Sarah', 'Amy', 'Sam'])
Sarah likes the color Yellow.
Amy likes the color Red.
Sam likes the color Blue.
{'Harry': 'Orange', 'Sarah': 'Purple', 'Amy': 'Red'}
```

Как видите, в словаре всегда есть пара «ключ — значение», разделенная двоеточием (:). Вместо использования индекса для доступа к отдельным значениям вы используете ключ. Специальная функция `keys()` позволяет получить список ключей, которыми можно манипулировать различными способами. Например, вы можете использовать ключи для итеративной обработки значений данных, содержащихся в словаре.



СОВЕТ

Словари немного похожи на отдельные таблицы в базе данных. Вы можете обновлять, добавлять и удалять записи в словаре, как показано. Функция `update()` может перезаписывать старые записи в словаре или добавлять туда новые.

§ 3. Работа с деревьями

Древовидная структура очень напоминает физический объект из природного мира. Использование деревьев помогает быстро организовать данные и находить их за меньшее время в сравнении со многими другими методами хранения данных. Деревья часто используются для поиска и сортировки, но у них есть и много других применений. Следующие

блоки помогут вам понять основы деревьев. Деревья встречаются во многих примерах в последующих главах.

Понимание основ деревьев

Построение дерева во многом похоже на создание дерева в физическом мире, если бы это было возможно. Каждый элемент, который вы добавляете в дерево, является *узлом*. Узлы соединяются друг с другом с помощью *связей*. Комбинация узлов и связей образует структуру, которая напоминает перевернутое дерево, как показано на рисунке 6.1.

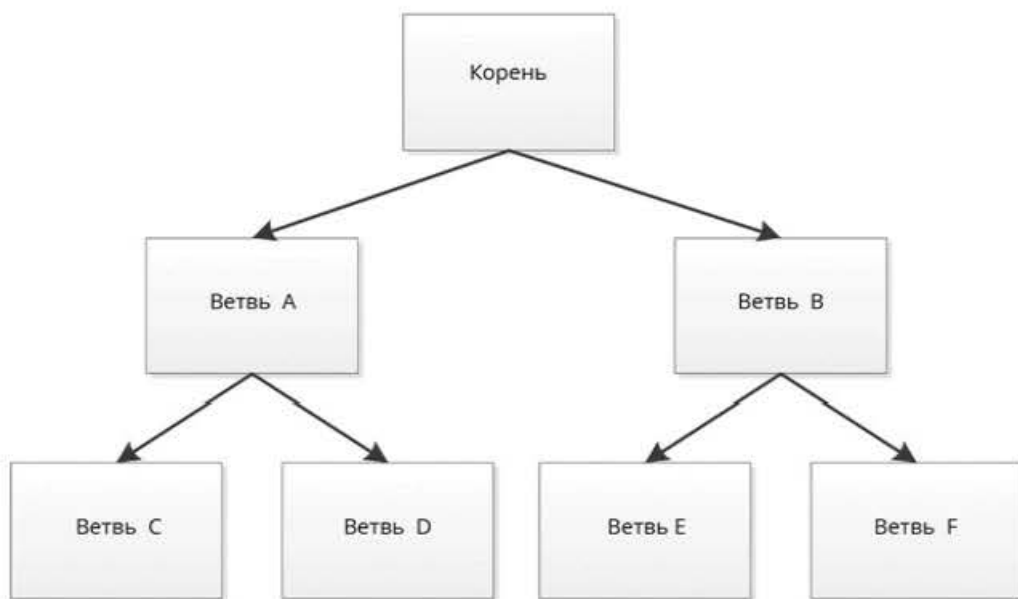


РИСУНОК 6.1.
Дерево
в Python очень
похоже на его
физический
аналог



ЗАПОМНИТЕ

Обратите внимание, что у дерева только один *корневой узел* — как и у физического дерева. Корневой узел служит отправной точкой для различных видов обработки, которые вы выполняете. К корневому узлу присоединены либо ветви, либо листья. *Листовой узел* всегда является конечной точкой дерева. *Узлы-ветви* поддерживают либо другие ветви, либо листья. Тип дерева, показанный на рисунке 6.1, является бинарным, поскольку у каждого узла не более двух соединений.

Глядя на дерево, видим, что ветвь *B* — потомок корневого узла. Это потому, что корневой узел появляется первым в списке. Лист *E* и лист *F* — потомки ветви *B*, что делает ее родителем листа *E* и листа *F*. Отношения между узлами важны, поскольку в обсуждениях деревьев часто рассматриваются отношения «потомок — родитель» между узлами. Без этих терминов обсуждения деревьев могли бы стать довольно запутанными.

Построение дерева

В Python нет встроенного объекта дерева. Базовая реализация дерева требует создания класса для хранения объекта данных дерева. Следующий код показывает, как можно создать базовый класс дерева:

```
class binaryTree:
    def __init__(self, nodeData, left=None, right=None):
        self.nodeData = nodeData
        self.left = left
        self.right = right
    def __str__(self):
        return str(self.nodeData)
```

Этот код просто создает базовый объект дерева, определяющий три элемента, которые должен включать узел: хранилище данных, левое соединение и правое соединение. Поскольку у листовых узлов нет соединений, значением по умолчанию для левого и правого соединения является `None`. Класс также включает метод для вывода содержимого `nodeData`, чтобы вы могли видеть, какие данные хранит узел.

При использовании такого простого дерева нужно хранить в `left` и `right` только ссылки на другие узлы. В противном случае код завершится с ошибкой, так как в нем нет обработки исключений. Поле `nodeData` может содержать любое значение. Следующий код показывает, как использовать класс `binaryTree` для построения дерева, проиллюстрированного на рисунке 6.1:

```
tree = binaryTree("Root")
BranchA = binaryTree("Branch A")
BranchB = binaryTree("Branch B")
tree.left = BranchA

tree.right = BranchB
LeafC = binaryTree("Leaf C")
LeafD = binaryTree("Leaf D")
LeafE = binaryTree("Leaf E")
LeafF = binaryTree("Leaf F")
BranchA.left = LeafC
BranchA.right = LeafD
BranchB.left = LeafE
BranchB.right = LeafF
```

Есть много способов построения дерева, но два распространенных метода — это построение сверху вниз (как показано в этом коде) или снизу вверх (когда сначала создаются листья). Конечно, на данном этапе вы не знаете, действительно ли дерево работает. *Обход дерева* означает проверку

связей и подтверждение того, что они действительно соединены так, как вы предполагаете. Следующий код показывает, как использовать рекурсию (описанную в главе 4) для обхода только что построенного дерева:

```
def traverse(tree):
    if tree.left != None:
        traverse(tree.left)
    if tree.right != None:
        traverse(tree.right)
    print(tree.nodeData)

traverse(tree)
```

Рекурсивная процедура `traverse()` начинает печать с листьев и движется к корню, поэтому вы видите следующий вывод:

```
Leaf C
Leaf D
Branch A
Leaf E
Leaf F
Branch B
Root
```

Как видно, `traverse()` печатает как листья, так и их родительский узел. Обход сначала следует по левой ветви, а затем по правой. Корневой узел выводится последним.



ПРИМЕЧАНИЕ

Существуют различные виды структур хранения данных. Вот краткий список наиболее распространенных структур:

- » **сбалансированные деревья** – вид дерева, которое поддерживает сбалансированную структуру путем реорганизации, что обеспечивает сокращение времени доступа;
- » **несбалансированные деревья** – дерево, которое размещает новые элементы данных в любом необходимом месте без учета баланса (такой метод добавления элементов ускоряет построение дерева, но снижает скорость доступа при поиске или сортировке);
- » **кучи** – сложная древовидная структура, позволяющая вставлять данные в структуру дерева (использование вставки данных ускоряет сортировку).

Далее в книге вы познакомитесь с алгоритмами, использующими сбалансированные деревья, несбалансированные деревья и кучи. Например, в главе 9 рассматривается алгоритм Дейкстры, а в главе 14 — кодирование Хаффмана.

§ 4. Представление отношений в виде графа

Графы — это еще одна распространенная структура данных, используемая в алгоритмах. Графы используются в таких областях, как карты для GPS-навигации, и во многих других случаях, где нисходящий подход деревьев не подходит. В следующих блоках графы рассматриваются подробнее.

Выходим за рамки деревьев

Граф можно рассматривать как расширение дерева. Как и в деревьях, в графах есть узлы, которые соединяются друг с другом для создания связей. Однако, в отличие от бинарных деревьев, у узла графа может быть более одного или двух соединений. На самом деле, узлы графа часто обладают множеством связей. Для простоты рассмотрим граф, показанный на рисунке 6.2.

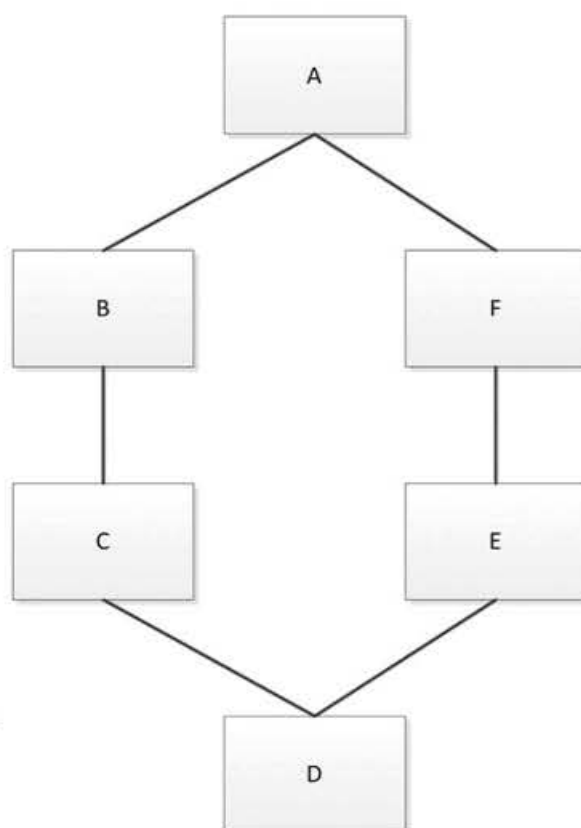


РИСУНОК 6.2.
Узлы графа могут соединяться друг с другом множеством способов

В данном случае граф образует кольцо, где узел *A* соединен с узлами *B* и *F*. Однако это необязательно должно быть так. Узел *A* мог бы быть изолированным узлом или также соединяться с узлом *C*. Граф показывает связность между узлами способом, который полезен для определения сложных отношений.

Графы также добавляют несколько новых особенностей, о которых вы, возможно, раньше не задумывались. Например, граф может включать концепцию направленности. В отличие от дерева, которое имеет отношения «родитель — потомок», узел графа может соединяться с любым другим узлом с учетом определенного направления. Подумайте об улицах в городе. Большинство улиц двунаправленные, но некоторые являются односторонними, допускающими движение только в одном направлении.

Визуальное представление связи в графе может не отражать реальных характеристик графа. Граф может назначать определенный вес конкретному соединению. Вес может определять расстояние между двумя точками, время, необходимое для прохождения маршрута, или предоставлять другую информацию.

Построение графов

Большинство разработчиков используют словари (или иногда списки) для построения графов. Использование словаря упрощает построение графа, поскольку ключом является имя узла, а значениями — связи для этого узла. Например, вот словарь, который создает граф, показанный на рисунке 6.2:

```
graph = {'A': ['B', 'F'],
         'B': ['A', 'C'],
         'C': ['B', 'D'],
         'D': ['C', 'E'],
         'E': ['D', 'F'],
         'F': ['E', 'A']}
```

Этот словарь отражает двунаправленную природу графа на рисунке 6.2. Он с таким же успехом мог бы определять однонаправленные связи или содержать узлы без каких-либо связей. Тем не менее словарь отлично подходит для этой цели, и вы увидите его использование в других местах книги. Теперь пришло время обойти граф, используя следующий код:

```
def find_path(graph, start, end, path=[]):
    path = path + [start]
```



```

        if start == end:
            print("Ending")
            return path

        for node in graph[start]:
            print("Checking Node ", node)

            if node not in path:
                print("Path so far ", path)

                newp = find_path(graph, node, end, path)
                if newp:
                    return newp

    find_path(graph, 'B', 'E')

```

Когда вы запускаете этот код, пример начинается с узла 'A' и проходит через все связи графа, создавая следующий вывод:

```

Checking Node A
Path so far ['B']
Checking Node B
Checking Node F
Path so far ['B', 'A']
Checking Node E
Path so far ['B', 'A', 'F']
Ending

['B', 'A', 'F', 'E']

```

В последующих главах обсуждается, как найти кратчайший путь. Пока что код находит просто какой-то путь. Он начинает с построения пути узел за узлом. Как и во всех рекурсивных процедурах, здесь требуется стратегия выхода — когда начальное значение совпадает с конечным, путь заканчивается.

Поскольку каждый узел в графе может соединяться с несколькими узлами, необходим цикл `for` для проверки каждого потенциального соединения. Когда рассматриваемый узел уже присутствует в пути, код пропускает его (что позволяет избежать бесконечного цикла — риска, характерного для алгоритмов на графах). В противном случае код отслеживает текущий путь и рекурсивно вызывает `find_path()` для поиска следующего узла в пути.

В ЭТОЙ ГЛАВЕ

- » Сортировка с использованием алгоритмов сортировки слиянием и быстрой сортировки
- » Поиск с использованием деревьев и кучи
- » Рассмотрение применения хеширования и словарей

ГЛАВА 7

Упорядочивание и поиск данных

Глава посвящена использованию четырех операций с данными: создания, чтения, обновления и удаления (create, read, update, and delete — CRUD) — для управления данными. Первый параграф главы фокусируется на сортировке данных. Размещение данных в порядке, облегчающем выполнение CRUD-операций, важно, поскольку чем меньше кода требуется для обеспечения доступа к данным, тем лучше. Кроме того, хотя сортировка данных может показаться не особенно важной, отсортированные данные делают поиск значительно быстрее, если способ сортировки соответствует поиску. Сортировка и поиск идут рука об руку: вы сортируете данные так, чтобы ускорить поиск.

Второй параграф главы посвящен поиску. Вы не удивитесь, узнав, что есть много различных способов поиска данных. Некоторые из этих методов медленнее других; у некоторых есть особенности, делающие их привлекательными для разработчиков. Дело в том, что идеальной стратегии поиска не существует, но поиски такого метода продолжаются.

В заключительном параграфе главы рассматриваются хеширование и словари. Использование индексации делает сортировку и поиск значительно быстрее, но также имеет компромиссы, которые нужно учитывать (например, использование дополнительных ресурсов). *Индекс* — это своего рода указатель или адрес. Это не сами данные, а указатель на них, подобно тому, как ваш адрес указывает на ваш дом. Поквартальный ручной

поиск вашего дома в городе был бы трудоемким, поскольку ищущему пришлось бы спрашивать каждого человека по каждому адресу, живете ли вы там, но найти ваш адрес в телефонной книге, а затем использовать этот адрес для поиска вашего дома намного быстрее.



ЗАПОМНИТЕ

Вам не нужно вводить исходный код для этой главы вручную. На самом деле, использовать загружаемый исходный код намного проще. Исходный код для этой главы можно найти в файлах `A4D2E; 07; Hashing.ipynb`, `A4D2E; 07; Search Techniques.ipynb` и `A4D2E; 07; Sorting Techniques.ipynb` из загружаемых исходников. Подробнее о том, как найти эти исходные файлы, смотрите во введении.

§ 1. Сортировка данных с помощью сортировки слиянием и быстрой сортировки

Сортировка — одна из важнейших операций при работе с данными. За прошедшие годы множество людей предложили различные способы сортировки данных. Все эти методы приводят к упорядоченным данным, но некоторые работают лучше других, а некоторые особенно хорошо подходят для конкретных задач. Следующие блоки помогут вам понять необходимость поиска и рассмотреть различные его варианты.

Почему важна сортировка данных

Можно привести аргументы против сортировки данных. В конце концов, данные остаются доступными, даже если их не сортировать, а сортировка требует времени. Конечно, проблема с несортированными данными такая же, как с ящиком для мелочей на вашей кухне (или где бы ни находился ваш ящик для мелочей — если вы вообще можете его найти). Поиск чего-либо в таком ящике отнимает много времени, потому что вы даже не можете предположить, где что находится. Вместо того чтобы просто протянуть руку и взять нужное, приходится перебирать множество ненужных предметов в попытках найти одну нужную вещь. К сожалению, нужного предмета может вообще не оказаться в ящике для мелочей; возможно, вы его выбросили или положили в другой ящик.

Ящик для мелочей в вашем доме подобен несортированным данным в вашей системе. Когда данные не отсортированы, приходится просматри-

вать элементы по одному, и вы даже не знаете, найдете ли нужное, пока не проверите каждый элемент в наборе данных. Это утомительный способ работы с данными. Пример бинарного поиска в блоке «Рассматриваем метод „разделяй и властвуй“» главы 4 хорошо показывает необходимость сортировки. Представьте, что пытаетесь найти элемент в списке, не отсортировав его сначала. Каждый поиск превращается в трудоемкий последовательный перебор.



ЗАПОМНИТЕ

Просто отсортировать данные недостаточно. Если у вас есть база данных сотрудников, отсортированная по фамилиям, но нужно найти сотрудника по дате рождения, такая сортировка бесполезна. Допустим, вы хотите найти всех сотрудников, у которых день рождения в определенный день. Чтобы найти нужную дату рождения, вам все равно придется просматривать весь набор данных по одному элементу. Следовательно, сортировка должна быть нацелена на конкретную потребность. Да, в одном случае вам нужна была база данных сотрудников, отсортированная по отделам, в другом — по фамилиям, а теперь для эффективного использования набора данных требуется сортировка по датам рождения.

Необходимость поддерживать несколько вариантов сортировки одних и тех же данных — причина, по которой разработчики создали индексы. Сортировка небольшого индекса выполняется быстрее, чем сортировка всего набора данных. Индекс поддерживает определенный порядок данных и указывает на полный набор данных, что позволяет находить нужную информацию предельно быстро. Поддерживая индекс для каждого требования к сортировке, можно эффективно сократить время доступа к данным и позволить нескольким пользователям одновременно обращаться к данным в нужном им порядке. В параграфе «Опираясь на хеширование» далее в этой главе вы узнаете, как работает индексация и почему она действительно необходима в некоторых случаях, несмотря на дополнительное время и ресурсы, требуемые для поддержания индексов.



ПРИМЕЧАНИЕ

Есть много способов классификации алгоритмов сортировки. Один из них — по скорости сортировки. При оценке эффективности конкретного алгоритма сортировки в тестах производительности обычно учитывают два фактора:

- » **сравнений.** Чтобы переместить данные из одного места в наборе данных в другое, нужно знать, куда их перемещать, что означает сравнение целевых данных с другими данными в наборе. Меньшее количество сравнений означает лучшую производительность;
- » **обменов.** В зависимости от того, как написан алгоритм, данные могут не попасть в свое конечное положение в наборе данных с первой попытки. Данные могут фактически

перемещаться несколько раз. Количество обменов существенно влияет на скорость, поскольку в этом случае вы действительно перемещаете данные из одного места в другое в памяти. Меньшее количество и меньший размер обменов (например, при использовании индексов) означает лучшую производительность.

Наивное упорядочивание данных с целью облегчения поиска (и других задач) означает их упорядочивание методом грубой силы — без каких-либо попыток предугадать, где данные должны появиться в списке. Кроме того, эти методы обычно работают со всем набором данных вместо применения подходов, которые могли бы сократить время сортировки (таких, как метод «разделяй и властвуй», описанный в главе 4). Однако эти алгоритмы поиска также относительно просты для понимания и эффективно используют ресурсы, поэтому не стоит полностью их исключать. Хотя в эту категорию попадает множество алгоритмов поиска, в следующих блоках рассматриваются два наиболее популярных подхода.

Использование сортировки выбором

Сортировка выбором заменила своего предшественника — сортировку пузырьком, поскольку она обычно обеспечивает лучшую производительность. Хотя у обоих алгоритмов наихудшая скорость сортировки $O(n^2)$, сортировка выбором выполняет меньше обменов. Сортировка выбором работает одним из двух способов: либо ищет наименьший элемент в списке и помещает его в начало списка (гарантируя, что элемент находится в правильном месте), либо ищет наибольший элемент и помещает его в конец списка. В любом случае эта сортировка исключительно проста в реализации и гарантирует, что элементы сразу оказываются на своих окончательных позициях после перемещения (поэтому некоторые называют ее сортировкой на месте методом сравнения). Вот пример сортировки выбором. (Этот код можно найти в файле исходного кода **A4D2E; 07; Sorting Techniques.ipynb**; подробнее см. во введении.)

```
data = [9, 5, 7, 4, 2, 8, 1, 10, 6, 3]

for scanIndex in range(0, len(data)):
    minIndex = scanIndex

    for compIndex in range(scanIndex + 1, len(data)):
        if data[compIndex] < data[minIndex]:
            minIndex = compIndex

    if minIndex != scanIndex:
        data[scanIndex], data[minIndex] = \
```



```
data[minIndex], data[scanIndex]
print(data)
```

При выполнении кода вы увидите следующий результат (обратите внимание, как каждое число оказывается на правильной позиции по мере выполнения сортировки):

```
[1, 5, 7, 4, 2, 8, 9, 10, 6, 3]
[1, 2, 7, 4, 5, 8, 9, 10, 6, 3]
[1, 2, 3, 4, 5, 8, 9, 10, 6, 7]
[1, 2, 3, 4, 5, 6, 9, 10, 8, 7]
[1, 2, 3, 4, 5, 6, 7, 10, 8, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 10, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Переходим к сортировке вставками

Сортировка вставками работает, используя один элемент в качестве начальной точки и добавляя элементы слева или справа от него в зависимости от того, меньше или больше они выбранного элемента. По мере увеличения количества отсортированных элементов алгоритм сравнивает новые элементы с уже отсортированными и вставляет новый элемент в правильную позицию в списке. Сортировка вставками имеет лучшую временную сложность $O(n)$ и худшую временную сложность $O(n^2)$.



ПРИМЕЧАНИЕ

Пример лучшего случая — когда весь набор данных уже отсортирован, поскольку сортировке вставками не придется перемещать никакие значения. Пример худшего случая — когда весь набор данных отсортирован в обратном порядке, поскольку каждая вставка потребует перемещения всех значений, которые уже находятся в выходном массиве. Подробнее о математике, лежащей в основе такой сортировки, можно прочитать на <https://www.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/analysis-of-insertion-sort>.

Сортировка вставками по-прежнему остается методом грубой силы для сортировки элементов, но может требовать меньше сравнений, чем сортировка выбором. Вот пример сортировки вставками:

```
data = [9, 5, 7, 4, 2, 8, 1, 10, 6, 3]

for scanIndex in range(1, len(data)):
    temp = data[scanIndex]

    minIndex = scanIndex
```



```
while minIndex > 0 and temp < data[minIndex - 1]:  
    data[minIndex] = data[minIndex - 1]  
    minIndex -= 1  
  
data[minIndex] = temp  
print(data)
```

Результат примера показывает, что для этого случая требуется девять итераций, в отличие от семи в предыдущем примере. Однако в других случаях сортировка вставками работает лучше, чем сортировка выбором. Все зависит от входных данных.

```
[5, 9, 7, 4, 2, 8, 1, 10, 6, 3]  
[5, 7, 9, 4, 2, 8, 1, 10, 6, 3]  
[4, 5, 7, 9, 2, 8, 1, 10, 6, 3]  
[2, 4, 5, 7, 9, 8, 1, 10, 6, 3]  
[2, 4, 5, 7, 8, 9, 1, 10, 6, 3]  
[1, 2, 4, 5, 7, 8, 9, 10, 6, 3]  
[1, 2, 4, 5, 7, 8, 9, 10, 6, 3]  
[1, 2, 4, 5, 6, 7, 8, 9, 10, 3]  
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Применение улучшенных методов сортировки

По мере совершенствования технологий сортировки алгоритмы начинают использовать более интеллектуальный подход к упорядочиванию данных. Идея заключается в том, чтобы сделать задачу меньше и проще в управлении. Вместо работы с целым набором данных умные алгоритмы сортировки работают с отдельными элементами, уменьшая объем работы, необходимый для выполнения задачи. В следующих блоках рассматриваются два таких интеллектуальных метода сортировки.

Перестановка данных с помощью сортировки слиянием

Сортировка слиянием работает по принципу «разделяй и властвуй». Она начинается с разбиения набора данных на отдельные элементы и с их сортировки. Затем происходит слияние этих элементов так, чтобы объединенная часть оставалась отсортированной. Процесс сортировки и слияния продолжается до тех пор, пока весь набор данных снова не станет единым целым. В худшем случае скорость сортировки слиянием составляет $O(n \log n)$, что делает ее значительно быстрее методов, рассмотренных в предыдущем блоке (поскольку $\log n$ всегда меньше n). Для этой сортировки фактически требуются две функции. Первая функция рекурсивно разделяет части и соединяет их обратно.

```
data = [9, 5, 7, 4, 2, 8, 1, 10, 6, 3]

def mergeSort(list):
    # Determine whether the list is broken into
    # individual pieces.
    if len(list) < 2:
        return list

    # Find the middle of the list.
    middle = len(list)//2
    # Break the list into two pieces.
    left = mergeSort(list[:middle])
    right = mergeSort(list[middle:])

    # Merge the two sorted pieces into a larger piece.
    print("Left side: ", left)
    print("Right side: ", right)
    merged = merge(left, right)
    print("Merged ", merged)
    return merged
```

Вторая функция выполняет непосредственно задачу слияния двух частей с помощью итеративного процесса. Вот код, используемый для объединения двух частей:

```
def merge(left, right):
    # When the left side or the right side is empty,
    # it means that this is an individual item and is
    # already sorted.
    if not len(left):
        return left
    if not len(right):
        return right
```

```

        return right

    # Define variables used to merge the two pieces.
    result = []
    leftIndex = 0
    rightIndex = 0
    totalLen = len(left) + len(right)

    # Keep working until all of the items are merged.
    while (len(result) < totalLen):

        # Perform the required comparisons and merge
        # the pieces according to value.
        if left[leftIndex] < right[rightIndex]:
            result.append(left[leftIndex])
            leftIndex+= 1
        else:
            result.append(right[rightIndex])
            rightIndex+= 1

        # When the left side or the right side is longer,
        # add the remaining elements to the result.
        if leftIndex == len(left) or \
            rightIndex == len(right):
            result.extend(left[leftIndex:]
                          or right[rightIndex:])
            break

    return result

mergeSort(data)

```

Операторы печати в коде помогают увидеть, как работает процесс слияния. Хотя процесс кажется довольно сложным, он на самом деле относительно прост, если проследить за процессом слияния, показанным здесь:

```

Left side: [9]
Right side: [5]
Merged [5, 9]
Left side: [4]
Right side: [2]
Merged [2, 4]
Left side: [7]
Right side: [2, 4]
Merged [2, 4, 7]
Left side: [5, 9]

```



```
Right side: [2, 4, 7]
Merged [2, 4, 5, 7, 9]
Left side: [8]
Right side: [1]
Merged [1, 8]
Left side: [6]
Right side: [3]
Merged [3, 6]
Left side: [10]
Right side: [3, 6]
Merged [3, 6, 10]
Left side: [1, 8]
Right side: [3, 6, 10]
Merged [1, 3, 6, 8, 10]
Left side: [2, 4, 5, 7, 9]
Right side: [1, 3, 6, 8, 10]
Merged [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Решение проблем сортировки наилучшим способом с помощью быстрой сортировки

Быстрая сортировка — один из самых быстрых методов сортировки данных. Изучая в интернете информацию о сортировке слиянием и быстрой сортировке, можно заметить, что некоторые люди предпочитают использовать один метод вместо другого в определенных ситуациях. Например, большинство считает, что быстрая сортировка лучше всего подходит для сортировки массивов, а сортировка слиянием — для сортировки связанных списков (см. обсуждение на <https://www.geeksforgeeks.org/why-quick-sort-preferred-for-arrays-and-merge-sort-for-linked-lists>). Тони Хоар написал первую версию быстрой сортировки в 1959 году, но с тех пор разработчики создали множество других версий этого алгоритма. Среднее время сортировки для быстрой сортировки составляет $O(n \log n)$, однако в худшем случае время сортировки достигает $O(n^2)$.

АНАЛИЗ НАИХУДШЕЙ ПРОИЗВОДИТЕЛЬНОСТИ БЫСТРОЙ СОРТИРОВКИ

Быстрая сортировка редко демонстрирует наихудшее время выполнения. Тем не менее даже у модифицированных версий быстрой сортировки может быть время выполнения $O(n^2)$ в случаях, когда:

- набор данных уже отсортирован в нужном порядке,
- набор данных отсортирован в обратном порядке,
- все элементы в наборе данных одинаковы.

Все эти проблемы возникают из-за опорного элемента, который использует менее интеллектуальная функция сортировки (часто выбор оказывается неудачным). К счастью, правильный подход к программированию может смягчить эти проблемы, определяя в качестве опорного элемента что-то иное, чем крайний левый или правый индекс. Современные версии быстрой сортировки используют следующие методы:

- выбор случайного индекса;
- выбор среднего индекса раздела;
- выбор медианы первого, среднего и последнего элемента раздела в качестве опорного элемента (особенно для длинных разделов).

Первая часть задачи заключается в разбиении данных. Код выбирает опорную точку, которая определяет левую и правую стороны сортировки. Вот код разбиения для этого примера:

```
data = [9, 5, 7, 4, 2, 8, 1, 10, 6, 3]

def partition(data, left, right):
    pivot = data[left]
    lIndex = left + 1
    rIndex = right

    while True:
        while lIndex <= rIndex and data[lIndex] <= pivot:
            lIndex += 1
        while rIndex >= lIndex and data[rIndex] >= pivot:
            rIndex -= 1
        if rIndex <= lIndex:
            break
```

```

        data[lIndex], data[rIndex] = \
            data[rIndex], data[lIndex]
        print(data)

    data[left], data[rIndex] = data[rIndex], data[left]
    print(data)
    return rIndex

```

Внутренний цикл этого примера постоянно ищет элементы, находящиеся не на своих местах, и меняет их местами. Когда код больше не может менять элементы местами, он выходит из цикла и устанавливает новую опорную точку, которую возвращает вызывающей функции. Это итеративная часть процесса. Рекурсивная часть процесса обрабатывает левую и правую части набора данных, как показано здесь:

```

def quickSort(data, left, right):
    if right <= left:
        return
    else:
        pivot = partition(data, left, right)
        quickSort(data, left, pivot-1)
        quickSort(data, pivot+1, right)

    return data

quickSort(data, 0, len(data)-1)

```

Количество сравнений и обменов для этого примера относительно невелико в сравнении с другими примерами. Вот вывод из этого примера:

```

[9, 5, 7, 4, 2, 8, 1, 3, 6, 10]
[6, 5, 7, 4, 2, 8, 1, 3, 9, 10]
[6, 5, 3, 4, 2, 8, 1, 7, 9, 10]
[6, 5, 3, 4, 2, 1, 8, 7, 9, 10]
[1, 5, 3, 4, 2, 6, 8, 7, 9, 10]
[1, 5, 3, 4, 2, 6, 8, 7, 9, 10]
[1, 2, 3, 4, 5, 6, 8, 7, 9, 10]
[1, 2, 3, 4, 5, 6, 8, 7, 9, 10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

```


§ 2. Использование деревьев поиска и кучи

Деревья поиска позволяют быстро искать данные. В главе 4 вы познакомились с концепцией бинарного поиска, а параграф «Работа с деревьями» главы 6 помог вам в некоторой степени понять деревья. Получение элементов данных, размещение их в отсортированном порядке в дереве и последующий поиск по этому дереву — один из самых быстрых способов поиска информации.

Особый вид древовидной структуры — *бинарная куча*, в которой все узлы размещаются в определенном порядке. Корневой узел всегда содержит наименьшее значение. При рассмотрении ветвей видно, что значения в узлах верхних уровней всегда меньше значений в узлах нижних уровней и листьях. Такой порядок обеспечивает сбалансированность дерева и его предсказуемую структуру, что делает поиск крайне эффективным. Платой за это — необходимость поддержания сбалансированности дерева. В следующих блоках подробно описывается, как работают деревья поиска и куча.

Рассматриваем необходимость эффективного поиска

Из всех задач, которые выполняют приложения, поиск — самый времязатратный и одновременно самый востребованный. Хотя добавление данных (и их последующая сортировка) тоже требует времени, польза от создания и поддержания набора данных проявляется при его использовании для выполнения полезной работы, что подразумевает поиск в нем важной информации. Следовательно, иногда можно мириться с менее эффективными операциями CRUD и даже с не самым оптимальным алгоритмом сортировки, но поиск должен выполняться максимально эффективно. Единственная проблема заключается в том, что нет универсального метода поиска, который бы выполнял все задачи с абсолютной эффективностью, поэтому нужно взвешивать варианты, исходя из того, что вы планируете делать в рамках поисковых процедур.

Два наиболее эффективных метода поиска основаны на использовании бинарного дерева поиска (BST) и бинарной кучи. Оба метода поиска опираются на древовидную структуру для хранения ключей, используемых для доступа к элементам данных. Однако организация этих двух методов различается, что и объясняет преимущества одного над другим при выполнении определенных задач. На рисунке 7.1 показана организация BST.

Обратите внимание, как ключи следуют порядку, при котором меньшие числа располагаются слева, а большие — справа. Корневой узел содержит значение, находящееся в середине диапазона ключей, что обеспечивает BST понятный, хорошо сбалансированный подход к хранению ключей. Сравните эту организацию с бинарной кучей, показанной на рисунке 7.2.

РИСУНОК 7.1.
Расположение
ключей при
использовании
BST

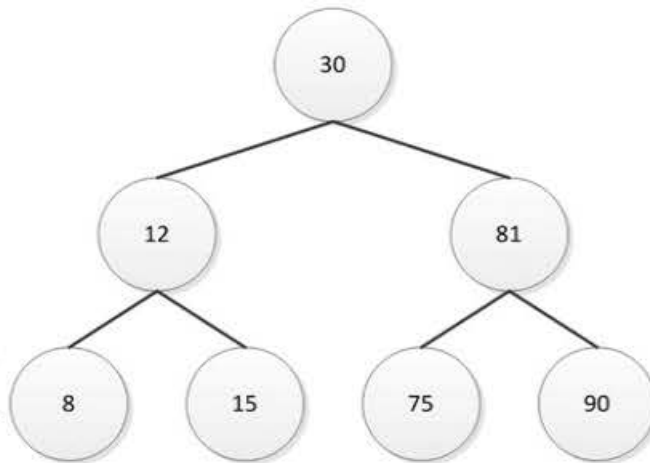
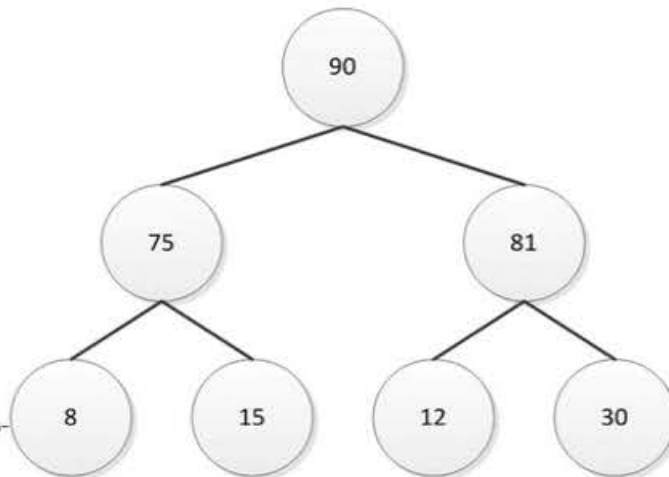


РИСУНОК 7.2.
Расположение
ключей в бинар-
ной куче



Каждый уровень содержит значения, которые меньше значений предыдущего уровня, а корень содержит максимальное значение ключа для дерева. Кроме того, в данном конкретном случае меньшие значения располагаются слева, а большие — справа (хотя этот порядок не является строго обязательным). На рисунке фактически изображена *максимальная бинарная куча*. Можно также создать *минимальную бинарную кучу*, в которой корень содержит наименьшее значение ключа, а каждый следующий уровень содержит более высокие значения, причем самые высокие значения появляются в листьях.



ЗАПОМНИТЕ

Как отмечалось ранее, у бинарного дерева поиска есть преимущества перед бинарной кучей при выполнении поиска. Вот основные из этих преимуществ:

- » Поиск элемента требует времени $O(\log n)$, когда дерево сбалансировано, в отличие от $O(n)$ для бинарной кучи.
- » Вывод элементов по порядку требует всего $O(\log n)$ времени, в отличие от $O(n \log n)$ для бинарной кучи.
- » Поиск нижней и верхней границы требует времени $O(\log n)$.
- » Поиск K -го наименьшего или наибольшего элемента требует времени $O(\log n)$, когда дерево правильно настроено.

Важность этих временных показателей зависит от вашего приложения. Бинарное дерево поиска лучше всего работает в ситуациях, когда больше времени тратится на поиск и меньше — на построение дерева. Бинарная куча лучше подходит для динамических ситуаций, когда ключи регулярно меняются. У бинарной кучи тоже есть преимущества, как описано в следующем списке:

- » Создание необходимых структур требует меньше ресурсов, поскольку бинарные кучи основаны на массивах, что делает их более дружелюбными к кешу.
- » Построение бинарной кучи требует времени $O(n)$, в отличие от бинарного дерева поиска, которому требуется время $O(n \log n)$.
- » Нет необходимости использовать указатели для реализации дерева.
- » Использование вариаций бинарной кучи (например, кучи Фибоначчи) предлагает такие преимущества, как время увеличения и уменьшения ключа $O(1)$ (для выполнения таких задач, как добавление и удаление элементов).

Построение бинарного дерева поиска

Существует несколько способов построения бинарного дерева поиска (BST). Некоторые используют словарь, другие — собственный код (примеры можно найти в статьях по адресам: <https://interactivepython.org/courselib/static/pythonds/Trees/SearchTreeImplementation.html> и <https://code.activestate.com/recipes/577540-python-binary-search-tree/>). Однако большинство разработчиков не хотят заново изобретать

велосипед, когда дело касается BST. С учетом этого вам нужен такой пакет, как `bintrees`, который предоставляет весь необходимый функционал для создания BST и взаимодействия с ним с использованием минимума кода. Чтобы загрузить и установить `bintrees`, откройте командную строку, введите: `pip install bintrees` и нажмите Enter. После этого вы увидите, что `bintrees` установлен в вашей системе. Документация к этому пакету доступна по адресу <https://pypi.org/project/bintrees/2.2.0>.

Пакет `bintrees` можно использовать для различных задач, но в этом блоке мы рассмотрим конкретно BST. В данном случае дерево будет несбалансированным. Следующий код показывает, как построить и отобразить BST с помощью `bintrees`. (Этот код можно найти в загружаемом файле исходного кода `A4D2E: 07: Search Techniques.ipynb`; подробнее см. во введении.)

```
from bintrees import BinaryTree

data = {3:'White', 2:'Red', 1:'Green', 5:'Orange',
        4:'Yellow', 7:'Purple', 0:'Magenta'}

tree = BinaryTree(data)
tree.update({6:'Teal'})

def displayKeyValue(key, value):
    print('Key: ', key, 'Value: ', value)

tree.foreach(displayKeyValue)
print('Item 3 contains: ', tree.get(3))
print('The maximum item is: ', tree.max_item())
```

При запуске кода вы увидите следующий вывод:

```
Key: 0 Value: Magenta
Key: 1 Value: Green
Key: 2 Value: Red
Key: 3 Value: White
Key: 4 Value: Yellow
Key: 5 Value: Orange
Key: 6 Value: Teal
Key: 7 Value: Purple
Item 3 contains: White
The maximum item is: (7, 'Purple')
```

Для создания бинарного дерева нужно предоставить пары «ключ — значение». Один из способов выполнить эту задачу — создать словарь, как показано выше. После создания дерева вы можете использовать функцию `update()` для добавления новых записей. Записи должны включать пару «ключ — значение», как показано в примере.



СОВЕТ

В примере используется функция для выполнения задачи с данными в дереве. В этом случае функция просто выводит пары «ключ — значение», но вы могли бы использовать дерево как входные данные для алгоритма анализа (и других задач). Функция `displayKeyValue()` передается в качестве аргумента функции `foreach()`, которая выводит пары «ключ — значение». У вас есть доступ и ко множеству других возможностей, таких как использование `get` для получения одного элемента или `max_item()` — для получения максимального элемента, хранящегося в дереве.

Выполнение специализированного поиска с помощью бинарной кучи

Как и в случае с бинарным деревом поиска, есть много способов реализации бинарной кучи. Написание ее вручную или использование словаря работает хорошо, но использование готового пакета делает процесс значительно быстрее и надежнее. Пакет `heapq` поставляется вместе с Python, поэтому его даже не нужно устанавливать. Документацию по пакету можно найти по адресу <https://docs.python.org/3/library/heapq.html>. Следующий пример показывает, как создать бинарную кучу и выполнить в ней поиск с помощью `heapq`:

```
import heapq

data = {3:'White', 2:'Red', 1:'Green', 5:'Orange',
        4:'Yellow', 7:'Purple', 0:'Magenta'}

heap = []
for key, value in data.items():
    heapq.heappush(heap, (key, value))
heapq.heappush(heap, (6, 'Teal'))
heap.sort()

for item in heap:
    print('Key: ', item[0], 'Value: ', item[1])
print('Item 3 contains: ', heap[3][1])
print('The maximum item is: ', heapq.nlargest(1, heap))
```

При запуске этого кода получается следующий вывод:

```
Key: 0 Value: Magenta
Key: 1 Value: Green
Key: 2 Value: Red
Key: 3 Value: White
Key: 4 Value: Yellow
Key: 5 Value: Orange
Key: 6 Value: Teal
Key: 7 Value: Purple
Item 3 contains: White
The maximum item is: [(7, 'Purple')]
```

Пример кода выполняет те же задачи и выдает тот же результат, что и пример в предыдущем блоке, но в данном случае использует бинарную кучу. Набор данных тот же, что и раньше. Однако обратите внимание на различие в способе добавления данных в кучу с помощью `heappush()`. Кроме того, после добавления нового элемента нужно вызвать `sort()`, чтобы элементы отображались в отсортированном порядке. Манипулирование данными больше похоже на работу со списком, в отличие от словарного подхода, используемого в `bintrees`. Какой бы подход вы ни использовали, стоит выбрать вариант, хорошо работающий с приложением, которое вы хотите создать, и обеспечивающий максимально быстрое время поиска для выполняемых задач.

§ 3. Опираясь на хеширование

Основная проблема большинства алгоритмов сортировки в том, что они сортируют все данные в наборе. Когда набор небольшой, вы едва замечаете объем данных, которые алгоритм сортировки пытается переместить. Однако по мере увеличения набора данных их перемещение становится заметным, когда вы часами сидите, уставившись в экран. Способ обойти эту проблему — сортировать только ключевую информацию. *Ключ* — это идентифицирующие данные для конкретной записи. Когда вы работаете с записью сотрудника, имя или номер сотрудника обычно служит ключом для доступа ко всей остальной информации о сотруднике. Бессмысленно сортировать всю информацию о сотрудниках, когда вам нужно отсортировать только ключи, и именно для этого используется хеширование. При работе с такими структурами данных вы получаете значительное преимущество в скорости, сортируя меньший объем данных, представленный ключами, а не записями в целом.

Раскладываем все по корзинам

До сих пор алгоритмы поиска и сортировки, описанные в книге, работали путем выполнения серии сравнений, пока не находили нужное значение. Процесс сравнения замедляет работу алгоритмов, поскольку каждое сравнение требует времени.

Более умный способ выполнения задачи заключается в том, чтобы предсказать местоположение конкретного элемента данных в структуре данных (какой бы она ни была) до того, как начать его поиск. Именно это и делает *хеш-таблица* — предоставляет средства для создания индекса ключей, который указывает на отдельные элементы в структуре данных, позволяя алгоритму легко предсказать местоположение данных. Размещение ключей в индексе происходит с помощью *хеш-функции*, которая преобразует ключ в числовое значение. Это числовое значение служит индексом в хеш-таблице, а хеш-таблица предоставляет указатель на полную запись в наборе данных. Поскольку хеш-функция дает воспроизводимые результаты, вы можете предсказать местоположение требуемых данных. Во многих случаях хеш-таблица обеспечивает время поиска $O(1)$. Другими словами, для поиска данных требуется всего одно сравнение.



ЗАПОМНИТЕ

Хеш-таблица содержит определенное количество *ячеек*, которые можно рассматривать как корзины для хранения данных. Каждая ячейка может хранить один элемент данных. Соотношение количества заполненных ячеек и количества доступных ячеек называется *коэффициентом заполнения*. Когда коэффициент заполнения высок, вероятность *коллизий* (когда у двух элементов данных одинаковое хеш-значение) тоже возрастает. В следующем блоке обсуждается, как избежать коллизий, но пока вам достаточно знать, что они могут возникать.

Один из наиболее типичных методов вычисления хеш-значения для входных данных — получение остатка от деления значения на количество ячеек. Например, если хотите сохранить число 54 в хеш-таблице, содержащей 15 ячеек, хеш-значение будет равно 9. Следовательно, значение 54 помещается в ячейку 9 хеш-таблицы, где ячейки пронумерованы от 0 до 14. Реальная хеш-таблица будет содержать гораздо больше ячеек, но для целей этого блока 15 вполне подходит. После помещения элемента в хеш-ячейку вы можете использовать хеш-функцию повторно, чтобы найти его местоположение.



СОВЕТ

Теоретически, если у вас есть идеальная хеш-функция и бесконечное количество ячеек, каждое значение, которое вы передаете хеш-функции, будет давать уникальное значение. В некоторых случаях вычисление хеша может стать довольно сложным, чтобы обеспечить уникальные значения в большинстве случаев. Однако чем сложнее вычисление хеша,

тем меньше выгоды вы получаете от хеширования, поэтому лучше всего придерживаться простоты.

Хеширование может работать с различными структурами данных. Однако для демонстрации в следующем примере используется простой список для хранения исходных данных и второй список для хранения результирующего хеша. (Этот код можно найти в загружаемом файле исходного кода **A4D2E; 07; Hashing.ipynb**; подробнее см. во введении.)

```
data = [22, 40, 102, 105, 23, 31, 6, 5]
hash_table = [None] * 15
tblLen = len(hash_table)

def hash_function(value, table_size):
    return value % table_size

for value in data:
    hash_table[hash_function(value, tblLen)] = value

print(hash_table)
```

При выполнении этого кода получаются следующие хеш-значения:

```
[105, 31, None, None, None, 5, 6, 22, 23, None, 40, None,
102, None, None]
```

Чтобы снова найти определенное значение, достаточно пропустить его через `hash_function()`. Например, `print(hash_table[hash_function(102, tblLen)])` выведет **102** в качестве результата после нахождения соответствующей записи в `hash_table`. Поскольку в данном конкретном случае хеш-значения уникальны, `hash_function()` может находить нужные данные каждый раз.

Избегание коллизий

Проблема возникает, когда у двух элементов данных одинаковое хеш-значение. Если просто записать значение в хеш-таблицу, вторая запись перезапишет первую, что приведет к потере данных. *Коллизии* — использование одного и того же хеш-значения двумя разными значениями — требуют наличия определенной стратегии для их обработки. Конечно, лучшая стратегия — это изначально избежать коллизии.

Один из методов избежания коллизий — обеспечить достаточно большой размер хеш-таблицы. Поддержание низкого коэффициента

заполнения — ваша первая линия защиты от необходимости проявлять креативность в использовании хеш-таблицы. Однако даже с большой таблицей не всегда удастся избежать коллизий. Иногда потенциальный набор данных настолько велик, а используемый набор настолько мал, что избежать проблемы становится невозможно. Например, если в школе 400 детей и вы полагаетесь на их номера социального страхования для идентификации, коллизии неизбежны, поскольку никто не станет создавать хеш-таблицу с миллиардом записей для такого количества детей. Это привело бы к огромной трате памяти. Следовательно, хеш-функции может потребоваться использовать не только простой остаток от деления для создания хеш-значения. Вот некоторые методы, которые можно использовать для избежания коллизий:

- » **Частичные значения.** При работе с некоторыми типами информации часть этой информации повторяется, что может создавать коллизии. Например, первые три цифры телефонного номера могут повторяться для определенной области, поэтому удаление этих цифр и использование только оставшихся четырех может помочь решить проблему коллизий.
- » **Свертка.** Создание уникального числа может быть таким же простым, как разделение исходного числа на части, сложение этих частей и использование результата в качестве хеш-значения. Например, используя телефонный номер 555-1234, хеш можно начать с разбиения его на части: 55 51234, а затем сложить результаты вместе, чтобы получить 340 как число, используемое для генерации хеша.
- » **Метод средних квадратов.** Хеш-функция возводит значение в квадрат, использует определенное количество цифр из центра полученного числа и отбрасывает остальные цифры. Например, возьмем число 120. При возведении в квадрат получаем 14400. Хеш-функция может использовать 440 для генерации хеш-значения, отбросив 1 слева и 0 справа.

Очевидно, существует столько способов генерации хеша, сколько может придумать человеческое воображение. К сожалению, никакая креативность не решит все проблемы коллизий, которые по-прежнему вероятны. Поэтому нужен другой план. Когда происходит коллизия, можно использовать один из следующих методов для ее разрешения:

- » **Линейное пробирование (разновидность открытой адресации).** Код сохраняет значение в следующей свободной ячейке, последовательно просматривая ячейки, пока не найдет свободную. Проблема подхода в том, что он предполагает наличие свободной ячейки для каждого потенциального значения, что не всегда возможно. Кроме того, открытая адресация означает, что поиск значительно замедляется при увеличении коэффициента заполнения. Вы больше не можете найти нужное значение с первого сравнения.
- » **Рехеширование.** Код хеширует хеш-значение плюс константу. Например, рассмотрим значение 1020 при работе с хеш-таблицей, содержащей 30 ячеек и константу 100. В этом случае хеш-значение равно 22. Однако если ячейка 22 уже содержит значение, то рехеширование $((22 + 100) \% 30)$ дает новое хеш-значение 2. В этом случае не нужно последовательно искать значение в хеш-таблице. При правильной реализации поиск может по-прежнему включать небольшое количество сравнений для нахождения целевого значения.
- » **Метод цепочек.** Каждая ячейка в хеш-таблице может содержать несколько значений. Подход можно реализовать, используя список внутри списка. Каждый раз при возникновении коллизии код просто добавляет значение в список в целевой ячейке. Преимущество подхода в том, что хеш всегда будет указывать на правильную ячейку, но список внутри этой ячейки все равно потребует последовательного (или иного) поиска для нахождения конкретного значения.

Создание собственной хеш-функции

Иногда может потребоваться создать пользовательские хеш-функции, чтобы ответить потребностям используемого алгоритма или улучшить его производительность. Помимо криптографических применений (которым можно посвятить отдельную книгу), в главе 12 представлены распространенные алгоритмы, использующие различные хеш-функции, такие как фильтр Блума, HyperLogLog и скетч с подсчетом минимумов. Эти алгоритмы используют свойства специальных хеш-функций для извлечения информации из огромных объемов данных.

ОБНАРУЖЕНИЕ НЕОЖИДАННЫХ СПОСОБОВ ИСПОЛЬЗОВАНИЯ ХЕШЕЙ

Помимо алгоритмов, подробно рассмотренных в книге, существуют и другие важные алгоритмы, основанные на хешах. Например, алгоритм локально-чувствительного хеширования (Locality-sensitive Hashing, LSH) использует большое количество хеш-функций для объединения, казалось бы, разрозненной информации. Если вы когда-либо задумывались, как маркетинговые компании и службы разведки связывают между собой различные куски информации на основе имен и адресов, которые не совпадают полностью (например, предполагая, что «Los Angels», «Los Angles» и «Los Angeles» все относятся к Лос-Анджелесу), то ответ – LSH. LSH разбивает информацию на части и обрабатывает ее с помощью множества хеш-функций, в результате чего формируется специальный хеш-результат – адрес корзины (bucket), в которую помещаются похожие слова. Сам алгоритм LSH достаточно сложен в реализации, но вы можете ознакомиться с ним в материале от Массачусетского технологического института (MIT): <http://www.mit.edu/~andoni/LSH>.

Множество примеров различных хеш-функций можно найти в пакете Python `hashlib`. Библиотека `hashlib` содержит следующие алгоритмы:

- » **алгоритмы безопасного хеширования (SHA)** – включают SHA1, SHA224, SHA256, SHA384 и SHA512. Выпущенные Национальным институтом стандартов и технологий (NIST) в качестве Федерального стандарта обработки информации (FIPS), SHA обеспечивают поддержку защищенных приложений и протоколов;
- » **алгоритм MD5 от RSA**. Изначально разработанный для приложений безопасности, этот хеш превратился в популярный способ контрольного суммирования файлов. Контрольные суммы сводят файлы к единственному числу, которое позволяет определить, был ли файл изменен после создания хеша (позволяя убедиться, что загруженный файл не поврежден и не был изменен хакером). Чтобы проверить целостность файла, достаточно сравнить MD5-хеш вашей копии с оригинальным, предоставленным автором файла.

Однако при работе со сложными приложениями, использующими большой набор данных, можно комбинировать выходные данные нескольких хеш-функций. Просто сложите результаты различных выходных данных после умножения одного или нескольких из них. Сумма двух хеш-функций, обработанных таким образом, сохраняет качества исходных хеш-функций, хотя результат получается другим, а восстановить исходные элементы суммы невозможно. Используя такой подход, вы получаете новую хеш-функцию, которую можно использовать как свой секретный хеш-рецепт для алгоритмов и приложений.

Следующий фрагмент кода использует пакет `hashlib` и хеш-алгоритмы `md5` и `sha1`. Вы просто указываете число для умножения внутри хеш-суммы. (Поскольку чисел бесконечно много, у вас есть функция, способная создавать бесконечное количество хешей.)

```
from hashlib import md5, sha1

def hash_f(element, i, length):
    """ Function to create many hash functions """
    h1 = int(md5(element.encode('ascii')).hexdigest(),16)
    h2 = int(sha1(element.encode('ascii')).hexdigest(),16)
    return (h1 + i*h2) % length
```

Вот код для тестирования функции `hash_f()` с соответствующим выводом:

```
print (hash_f("CAT", 1, 10**5))
64018

print (hash_f("CAT", 2, 10**5))
43738
```



ЗАПОМНИТЕ

Если вы задаетесь вопросом, где еще можно найти примеры использования хеш-таблиц вокруг вас, обратите внимание на словари Python. Словари — это, по сути, хеш-таблицы, хотя у них есть умный способ обработки коллизий, и вы не потеряете свои данные из-за того, что два хешированных ключа случайно дадут одинаковый результат. То, что индекс словаря использует хеш, также служит причиной его скорости при проверке наличия ключа. Кроме того, использование хеша объясняет, почему нельзя использовать любой тип данных в качестве ключа. Выбранный ключ должен быть чем-то, что Python может преобразовать в хеш-результат. Списки, например, не хешируемы, потому что они изменяемы; вы можете изменять их, добавляя или удаляя элементы. Тем не менее, если преобразовать список в строку, его можно использовать в качестве ключа для словаря в Python.



**ИССЛЕДОВАНИЕ
МИРА ГРАФОВ**

В ЭТОМ РАЗДЕЛЕ

Рассмотрение графов и их применения

Взаимодействие с графами различными способами

Навигация по графам в соответствии с конкретными потребностями

Поиск и ранжирование веб-страниц

В ЭТОЙ ГЛАВЕ

- » Определение, почему сети важны
- » Демонстрация техники построения графов
- » Рассмотрение функций графов
- » Использование числовых форматов для представления графов

ГЛАВА 8

Основы теории графов

Графы — это структуры, состоящие из набора узлов (или вершин), соединенных между собой набором ребер или дуг (в зависимости от представления). Когда вы думаете о графе, представляйте структуру наподобие карты, где каждое место на карте — это узлы, а улицы — ребра. Такое представление отличается от древовидного, где каждый путь заканчивается листовым узлом. Дерево может выглядеть как организационная схема или семейная иерархия. Что особенно важно, древовидные структуры действительно похожи на деревья и имеют четкое начало и конец. Эта глава начинается с объяснения важности сетей, которые представляют собой разновидность графов и широко используются для самых разных целей.



ЗАПОМНИТЕ

Графы можно представлять различными способами, большинство из которых абстрактны. Если только вы не обладаете исключительной способностью визуализировать абстракции в уме (большинство людей не обладают), вам нужно знать, как нарисовать граф, чтобы действительно его увидеть. Люди полагаются на зрение, чтобы понять, как работают вещи. Процесс преобразования чисел, представляющих граф, в графическую визуализацию называется *построением графа*. Такие языки, как Python, отлично справляются с построением графов, поскольку это невероятно важная функция. Фактически возможность построения графов — одна из причин, почему в книге используется Python, а не другой язык, например C (который хорош для выполнения другого набора задач).

После того как вы научитесь визуализировать граф, вам нужно будет знать, что делать с его графическим представлением. В этой главе мы начнем с измерения функциональности графа. Вы будете выполнять такие задачи, как подсчет ребер и вершин для определения сложности графа. Визуальное представление графа позволяет легче выполнять и такие задачи, как вычисление центральности. В главе 9 вы будете развивать то, что узнали в этой главе.

Числовое представление графа важно, даже если оно затрудняет понимание графа. График нужен вам, но компьютер, на самом деле, не понимает график (хотя и нарисовал его для вас). Думайте о компьютере как о более абстрактном мыслителе. С учетом необходимости представить граф в форме, которой компьютер может манипулировать, в главе рассматриваются три метода представления графа в числовом формате: матрицы, разреженные представления и списки. У всех этих методов есть свои преимущества и недостатки, и вы будете использовать их определенным образом в последующих главах (начиная с главы 9). Существуют и другие способы представления графа в числовом формате, но эти три метода хорошо послужат вам для взаимодействия с компьютером.



ЗАПОМНИТЕ

Вам не нужно вручную набирать исходный код для этой главы. На самом деле, использование загружаемого исходного кода намного проще. Исходный код главы можно найти в файлах `A4D2E; 08; Draw Graph.ipynb`, `A4D2E; 08; Graph Centrality.ipynb`, `A4D2E; 08; Graph Conversion.ipynb` и `A4D2E; 08; Graph Measurements.ipynb` из загружаемых материалов. Подробнее о том, как найти эти файлы, смотрите во введении.

§ 1. Объяснение важности сетей

Сеть — это особый вид графа, в котором вершинам (узлам или точкам), ребрам (дугам или линиям) или и тем, и другим присваиваются имена. Присвоение имен элементам графа снижает уровень абстракции и облегчает понимание графа. Данные, которые моделирует граф, становятся реальными в сознании человека, хотя граф — это действительно абстракция реального мира, представленная в форме, понятной как людям, так и компьютерам, хотя и по-разному. Следующие блоки помогут вам лучше понять важность сетей, чтобы вы могли увидеть, как их использование в книге упрощает задачу понимания работы алгоритмов и того, какую пользу вы можете извлечь из их применения.

Рассматриваем суть графа

Графы представляются в виде упорядоченных пар $G = (V, E)$, где G — это граф, V — список вершин, а E — список ребер, соединяющих вершины. Ребро фактически представляет собой числовую пару, выражающую две соединяемые им вершины. Следовательно, если у вас есть две вершины, представляющие города — Хьюстон (которому соответствует 1) и Даллас (которому соответствует 2), и вы хотите соединить их дорогой, вы создаете ребро **Highway**, содержащее пару ссылок на вершины: **Highway** = [Houston, Dallas]. Граф будет выглядеть как $G = [(Houston, Dallas)]$, что просто говорит о том, что есть первая вершина, Хьюстон, с соединением с Далласом, второй вершиной. Используя порядок представления вершин, Хьюстон смежен с Далласом; другими словами, автомобиль выезжает из Хьюстона и въезжает в Даллас.

Графы бывают нескольких видов. *Неориентированный граф* (как показано на рисунке 8.1) — это граф, в котором порядок элементов ребра не имеет значения. Дорожная карта в большинстве случаев представляет собой неориентированный граф, поскольку движение по дороге возможно в обоих направлениях.

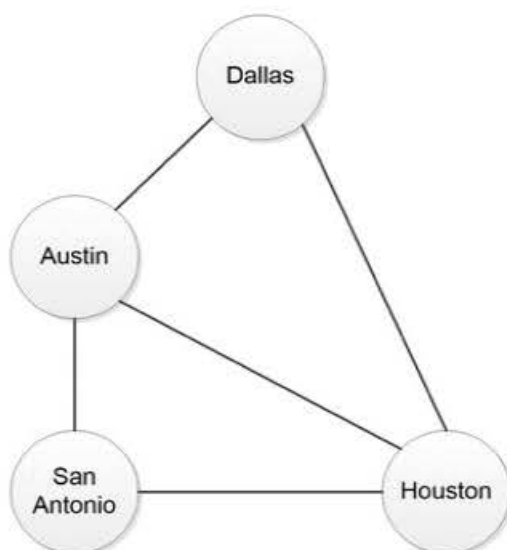
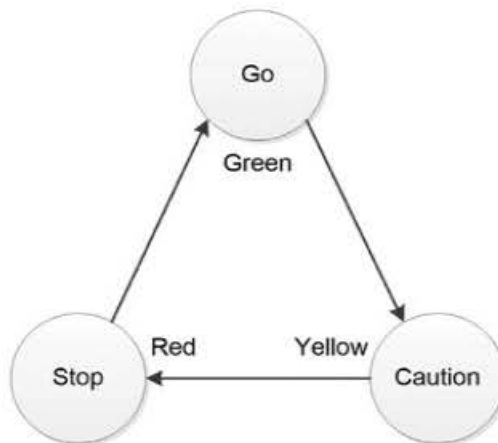


РИСУНОК 8.1.
Представление простого неориентированного графа

Ориентированный граф, как показано на рисунке 8.2, — это граф, в котором порядок записи ребер имеет значение, поскольку поток направлен от первого элемента ко второму. В этом случае ребра обычно называют *дугами*, чтобы отличить их от неориентированных ребер. Рассмотрим граф, представляющий последовательность сигналов светофора, где

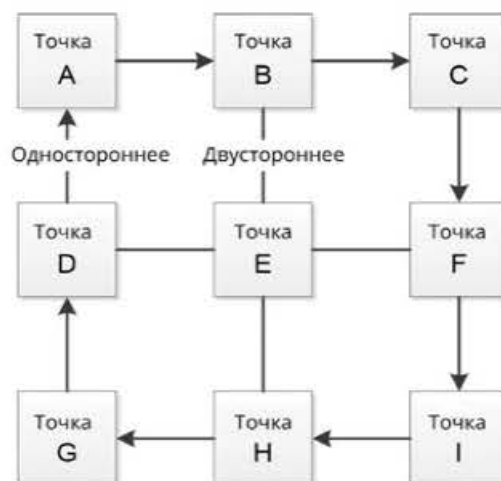
красный равен 1, желтый — 2, а зеленый — 3. Для выражения этой последовательности требуются три дуги: $Go = [Red, Green]$, $Caution = [Green, Yellow]$ и $Stop = [Yellow, Red]$. Порядок элементов важен, поскольку важна последовательность переходов от движения (Go) к вниманию ($Caution$) и стопу ($Stop$). Представьте хаос, который возник бы, если бы светофор игнорировал ориентированную природу этой последовательности.

РИСУНОК 8.2.
Создание ориентированной версии того же графа



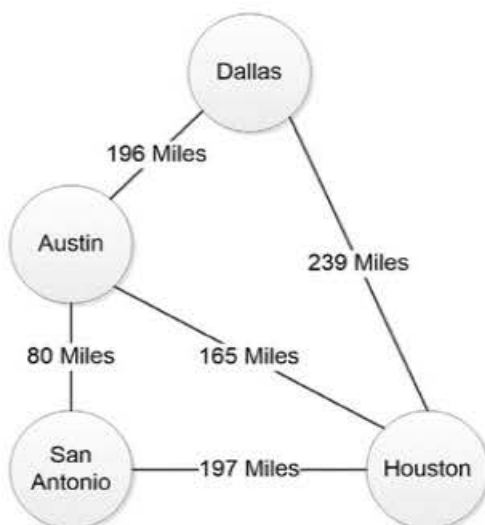
Третий важный тип графа, который нужно рассмотреть, — это *смешанный граф*. Подумайте снова о дорожной карте. Не всегда движение возможно в обоих направлениях на всех дорогах. При создании некоторых карт нужно учитывать наличие улиц с односторонним движением. На рисунке 8.3 видно, что попасть в точку A можно только из точки D . Однако из точки E можно двигаться в любом из четырех направлений, а в точку E можно попасть из точек B , D , F и H . Следовательно, вам нужны как неориентированные, так и ориентированные подграфы в одном графе — это то, что вы получаете в смешанном графе.

РИСУНОК 8.3.
Смешанный граф показывает сочетание ориентированных и неориентированных подграфов



Еще один тип графа, который стоит рассмотреть, — это *взвешенный граф* (показан на рисунке 8.4), в котором каждому ребру или дуге присвоены значения. Снова подумайте о дорожной карте. Людям часто нужно знать не только направление движения, но и насколько далеко находится следующий пункт назначения или сколько времени потребуется для того, чтобы туда добраться. Взвешенный граф предоставляет такую информацию, и веса можно использовать различными способами при выполнении вычислений с помощью графов.

РИСУНОК 8.4.
Использование
взвешенного
графа для
большей ре-
алистичности



Помимо взвешенного графа, при создании дорожной карты вам может понадобиться *граф с помеченными вершинами*. В графе с помеченными вершинами каждой вершине присвоено определенное имя. Представьте дорожную карту, где картограф не обозначил названия городов. Да, вы видите города, но без меток невозможно определить, какой из них есть какой. Дополнительные типы графов описаны на сайте <http://web.cecs.pdx.edu/~sheard/course/Cs163/Doc/Graphs.html>.

Графы повсюду

Графы могут показаться одной из тех эзотерических математических концепций, которые навевали на вас скуку в школе, но на самом деле они весьма увлекательны, поскольку мы постоянно используем их, даже не задумываясь об этом. Конечно, стоит отметить, что вам не придется иметь дело с числами, лежащими в основе графов. Возьмем, к примеру, карту. То, что вы видите, — это граф, но представленный в графическом формате, с городами, дорогами и многими другими элементами. Дело в том, что, глядя на карту, вы думаете именно о карте, а не о графе

(хотя ваш GPS-навигатор видит именно граф, поэтому он всегда может предложить кратчайший маршрут до пункта назначения). Если начать присматриваться, можно обнаружить множество привычных объектов, которые являются графами, хотя называются иначе.



ЗАПОМНИТЕ

У некоторых графов нет визуальной природы, но вы все равно не воспринимаете их как графы. Например, телефонные системы меню представляют собой форму ориентированного графа. На самом деле, несмотря на кажущуюся простоту, телефонные графы довольно сложны. Они могут включать циклы и множество других интересных структур. Попробуйте как-нибудь составить схему графа для какой-нибудь системы меню. Вы можете удивиться, насколько сложными некоторые из них могут быть.

Другой вид системы меню встречается в приложениях. Для выполнения задач большинство приложений проводит вас через серию шагов в особом виде подприложения, называемом *мастером*. Использование мастеров делает работу с внешне сложными приложениями намного проще, но чтобы создать работающий мастер, разработчик приложения должен создать граф, отображающий последовательность шагов.

Вас может удивить, но даже рецепты в кулинарных книгах представляют собой своего рода граф (и создание визуального представления связей между ингредиентами может оказаться весьма интересным). Каждый ингредиент в рецепте — это узел. Узлы соединяются ребрами, созданными инструкциями о смешивании ингредиентов. Конечно, рецепт — это просто разновидность химии, а химические графики показывают отношения между элементами в молекуле. (Да, люди действительно обсуждают это; вы можете увидеть одну из таких дискуссий на <https://stackoverflow.com/questions/7749073/representing-a-cooking-recipe-in-a-graph-database> и даже найти статью об этом на <https://medium.com/@condenastitaly/when-food-meets-ai-the-smart-recipe-project-eea259f53ed2>.)



ЗАПОМНИТЕ

Суть в том, что вы постоянно сталкиваетесь с этими графами, но не воспринимаете их как графы — вы видите их как что-то другое, например как рецепт или химическую формулу. Графы могут представлять множество различных отношений между объектами, подразумевая порядок последовательности, временную зависимость или причинно-следственные связи.

Социальная сторона графов

Графы имеют социальное значение, поскольку часто отражают отношения между людьми в различных ситуациях. Одно из самых очевидных применений графов — организационная структура. Подумайте: каждый узел представляет собой отдельного человека в организации, а ребра, соединяющие узлы, показывают различные взаимоотношения людей. То же самое верно и для других видов графов, например тех, что отображают семейную историю. Однако в первом случае граф неориентированный, поскольку коммуникация идет в обоих направлениях между руководителями и подчиненными (хотя характер общения различается в зависимости от направления). Во втором случае граф ориентированный, поскольку двое родителей производят детей на свет. Поток показывает направление наследования от основателя рода к нынешним детям.

Социальные сети тоже извлекают пользу из применения графов. Например, существует настоящая индустрия анализа взаимосвязей твитов в Twitter (на сайте <https://twittertoolsbook.com/10-awesome-twitter-analytics-visualization-tools> рассказывается лишь о некоторых из этих инструментов). Такой анализ опирается на использование графов для выявления связей между отдельными твитами.

Однако не нужно искать ничего сложнее, чем электронная почта, чтобы увидеть применение графов для социальных нужд. Корпус электронных писем компании Enron включает 200 399 сообщений от 158 высших руководителей — сообщений, выложенных в интернет Федеральной комиссией по регулированию энергетики (FERC). Ученые и исследователи использовали этот корпус для создания множества социальных графов, чтобы понять, как седьмая по величине компания в Соединенных Штатах оказалась вынуждена объявить о банкротстве в 2001 году (подробнее о том, как корпус помог и продолжает помогать развитию анализа сложных графов, можно узнать на сайтах <https://www.technologyreview.com/2013/07/02/177506/the-immortal-life-of-the-enron-e-mails> и <https://new.pythonforengineers.com/blog/analysing-the-enron-email-corpus>).

Даже на вашем компьютере есть социальные графы. Независимо от того, какое почтовое приложение вы используете, вы можете группировать письма различными способами, и эти методы группировки обычно опираются на графы для создания структуры. В конце концов, пытаться проследить ход обсуждения, не зная, какие сообщения являются ответами на другие сообщения, — безнадежное дело. Да, это возможно, но по мере увеличения количества сообщений такие попытки требуют все больше и больше времени, пока не становятся бессмысленными из-за временных ограничений, с которыми сталкивается большинство людей.

Понимание подграфов

Отношения, изображаемые графами, могут быть весьма сложными. Например, при отображении городских улиц многие из них допускают движение в обоих направлениях, что делает неориентированный граф идеальным способом представления. Однако некоторые улицы допускают движение только в одном направлении, что требует использования ориентированного графа. Сочетание двусторонних и односторонних улиц делает невозможным (или как минимум неудобным) представление с помощью графа одного типа. Объединение неориентированных и ориентированных графов в единый граф означает, что нужно создать подграфы для отображения каждого типа графа, а затем соединить эти подграфы в более крупный граф. Некоторые графы, содержащие подграфы, настолько распространены, что у них есть специальные названия — в данном случае это смешанный граф.

Подграфы полезны и для других целей. Например, может потребоваться проанализировать цикл внутри графа, что означает описание этого цикла как подграфа. При этом не нужен весь граф, достаточно только тех вершин и ребер, которые необходимы для проведения анализа. Такой подход используется во многих дисциплинах. Да, разработчики применяют его для проверки корректности работы частей приложения, но и городские инженеры используют его для понимания характера транспортных потоков в особо загруженных районах города. Медицинские специалисты тоже используют подграфы — для понимания движения крови или других жидкостей между органами в теле. Органы — это вершины, а кровеносные сосуды — ребра. Более того, многие из этих графов взвешенные — важно знать не только сам факт движения крови, но и ее количество.

Сложные графы могут также скрывать закономерности, о которых нужно знать. Например, один и тот же цикл может появляться в разных частях графа или вы можете увидеть одинаковый цикл в разных графах. Создав подграф из цикла, можно легко проводить сравнения внутри одного графа или между графами, чтобы увидеть, как они соотносятся. Например, биолог может захотеть сравнить цикл мутаций одного животного с циклом мутаций другого животного. Для такого сравнения биологу нужно будет создать представление в виде подграфа процессов для каждого животного целиком. (Интересный пример такого использования графов можно увидеть по адресу <https://www.sciencedirect.com/science/article/pii/S1359027896000569> — граф появляется в начале статьи как рисунок 1.)

§ 2. Определение того, как рисовать граф

Немногие люди способны визуализировать данные непосредственно в своем сознании. Однако большинству действительно необходимо графическое представление данных для их понимания. Это наглядно демонстрирует использование графиков в бизнес-презентациях. Вы могли бы рассказать другим о продажах за прошлый год, представив таблицы с цифрами. Через некоторое время большая часть вашей аудитории начала бы клевать носом, и вам никогда не удалось бы донести свою мысль. Причина проста: таблицы с цифрами точны и содержат много информации, но не представляют ее в той форме, которую люди понимают.



ЗАПОМНИТЕ

Визуализация данных и отображение показателей продаж в виде столбчатой диаграммы помогает людям легче увидеть взаимосвязи чисел. Если хотите подчеркнуть ежегодный рост продаж, столбчатая диаграмма с увеличивающимися столбцами наглядно это продемонстрирует. Любопытно, что использование графика фактически представляет данные менее точно. Попробуйте разглядеть на столбчатой диаграмме, что компания заработала 3 400 026,15 \$ в прошлом году и 3 552 215,82 \$ в этом году, практически невозможно. Да, таблица показала бы эту информацию, но людям не нужен такой уровень детализации — им просто нужно видеть ежегодный рост, то есть разницу в доходах от года к году. Однако компьютер интересуют именно детали, поэтому графики предназначены для людей, а матрицы — для компьютеров.

В следующих блоках вы познакомитесь с чудесами визуализации. Вы получите краткий обзор того, как работают графики в Python. (Эти принципы будут подробнее рассмотрены в последующих главах.) Представленные далее блоки дадут вам начальное понимание, чтобы вы могли легче разобраться в графиках, которые будут показаны позже.

Различаем ключевые атрибуты

Прежде чем рисовать граф, нужно знать о его атрибутах. Как упоминалось ранее, графы состоят из узлов (или вершин) и либо ребер (для неориентированных графов), либо дуг (для ориентированных графов). Любой граф, который вы захотите нарисовать, будет содержать эти элементы. Однако то, как вы представите эти элементы, частично зависит от выбранного вами пакета. Для простоты в книге используется комбинация двух пакетов:

- » **NetworkX** (<https://networkx.org>) – содержит код для рисования графов;
- » **matplotlib** (<http://matplotlib.org>) – предоставляет доступ ко всевозможным процедурам рисования (некоторые из них могут отображать графы, созданные с помощью NetworkX).



ЗАПОМНИТЕ

Чтобы использовать пакеты в Python, их нужно импортировать. Когда вам нужно использовать внешние пакеты, вы должны добавить специальный код, например следующие строки кода, которые обеспечивают доступ к **matplotlib** и **networkx**. (Вы можете найти этот код в загружаемом файле исходного кода **A4D2E; 08; Draw Graph.ipynb**; подробнее см. во введении.)

```
import networkx as nx
import matplotlib.pyplot as plt
```

Теперь, когда у вас есть доступ к пакетам, вы создаете граф. В данном случае граф — это своего рода контейнер, который содержит ключевые атрибуты, определяющие граф. Создание контейнера позволяет нарисовать граф, чтобы увидеть его позже. Следующий код создает объект **Graph** в **NetworkX**:

```
AGraph = nx.Graph()
```

Далее следует добавление ключевых атрибутов в **AGraph**. Вы должны добавить как узлы, так и ребра, используя следующий код:

```
Nodes = range(1,5)
Edges = [(1,2), (2,3), (3,4), (4,5), (1,3), (1,5)]
```

Как упоминалось ранее, **Edges** описывают связи между **Nodes**. В данном случае **Nodes** содержат значения от 1 до 5, поэтому **Edges** содержат связи между этими значениями.



API NETWORKX

ВНИМАНИЕ

Иногда разработчики вносят крупные, несовместимые изменения в прикладной программный интерфейс (API), чтобы улучшить продукт. В книге активно используется **NetworkX** для упрощения процесса создания графов. Однако версия **NetworkX**, используемая здесь, — новая, она несовместима со старым кодом. Чтобы работать с примерами из книги, у вас должен быть установлен **NetworkX** версии 2.5 или выше. Примеры были протестированы с использованием **NetworkX 2.6.2**, однако подойдет любая версия 2.5 или выше. Чтобы проверить свою версию **NetworkX**, используйте команду **pip list** или **conda list networkx**. **NetworkX** используется в главах 8, 9, 10, 11 и 16.

Конечно, **Nodes** и **Edges** сейчас просто существуют и не появятся как часть **AGraph**. Вы должны поместить их в контейнер, чтобы увидеть их. Используйте следующий код, чтобы добавить **Nodes** и **Edges** в **AGraph**:

```
AGraph.add_nodes_from(Nodes)
AGraph.add_edges_from(Edges)
```

Пакет **NetworkX** содержит множество функций для работы с отдельными узлами и ребрами, но показанный здесь подход — самый быстрый способ выполнения операций. Тем не менее вам может понадобиться добавить дополнительные ребра позже. Например, если хотите добавить ребро между узлами 2 и 4, вы можете вызвать функцию **AGraph.add_edge(2, 4)**.

Визуализация графа

Вы можете взаимодействовать с объектом-контейнером **AGraph**, созданным в предыдущем блоке, различными способами, но многие из них абстрактны и не очень удовлетворительны, если вы человек с визуальным мышлением. Иногда просто приятно увидеть содержимое объекта своими глазами. Следующий код отображает граф, содержащийся в **AGraph**:

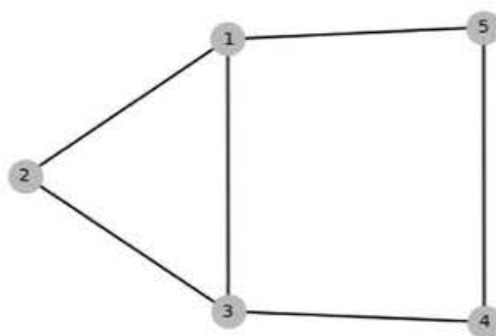
```
draw_params = {
    'with_labels': True,
    'node_color': 'skyblue',
    'node_size': 700, 'width': 2,
    'font_size': 14,
    'pos': nx.layout.spring_layout(AGraph, seed=1)}

nx.draw(AGraph, **draw_params)
plt.show()
```

Процесс отрисовки начинается с создания **draw_params**, который описывает, как рисовать граф. В этом примере мы указываем **NetworkX** использовать метки, установить небесно-голубой цвет узлов и различные другие параметры.

Функция **draw()** предоставляет различные аргументы, которые можно использовать для оформления визуализации. Однако вы можете передавать эти аргументы отдельно, используя **draw_params**, как показано в предыдущем коде. На рисунке 8.5 показан граф, содержащийся в **AGraph**.

РИСУНОК 8.5.
Визуальное
представле-
ние графа
облегчает его
понимание



§ 3. Измерение функциональности графа

Когда вы можете визуализировать и понять граф, нужно определить, какие его части важны. В конце концов, вы не хотите тратить время на анализ данных, которые не имеют большого значения в общей картине. Подумайте о специалисте, анализирующем транспортные потоки для улучшения дорожной системы. Перекрестки представляют собой вершины, а улицы — ребра, по которым движется транспорт. Зная, как движется транспорт, то есть какие вершины и ребра видят наибольший трафик, вы можете начать думать о том, какие дороги расширить и какие нуждаются в большем ремонте из-за более интенсивного движения.

Однако просто смотреть на отдельные улицы недостаточно. Новый небоскреб может привести к увеличению трафика, который затронет целый район. Небоскреб представляет собой центральную точку, вокруг которой транспортный поток становится важнее. Наиболее важные — вершины, центральные по отношению к новому небоскребу. Расчет *центральности* — наиболее важных вершин в графе — может помочь понять, какие части графа требуют большего внимания. В следующих блоках обсуждаются основные вопросы, которые нужно учитывать при измерении *функциональности графа* — способности графа моделировать конкретную задачу.

Подсчет ребер и вершин

По мере усложнения графов они передают больше информации, но еще становятся сложнее для понимания и манипулирования. Сложность графа определяется количеством его ребер и вершин. Однако для полного понимания структуры нужно рассматривать комбинацию ребер и вершин. Например, в графе может быть узел, который никак не связан с другими узлами. Создание такого узла допустимо — он может представлять значение, не имеющее связей с остальными. Используя приведенный ниже код, вы легко можете определить, что у узла 6 нет связей с другими, поскольку для него отсутствует информация о ребрах. (Этот код можно найти в загружаемом файле исходного кода **A4D2E: 08; Graph Measurements.ipynb**; подробнее о том, как найти этот файл, см. во введении.)

```
import networkx as nx
import matplotlib.pyplot as plt

AGraph = nx.Graph()

Nodes = range(1,5)
Edges = [(1,2), (2,3), (3,4), (4,5), (1,3), (1,5)]

AGraph.add_nodes_from(Nodes)
AGraph.add_edges_from(Edges)

AGraph.add_node(6)
sorted(nx.connected_components(AGraph))
```

При запуске этого кода вы увидите следующий результат:

```
[[1, 2, 3, 4, 5], {6}]
```

Вывод показывает, что узлы с 1-го по 5-й соединены между собой, а у узла 6 нет соединений. Эту ситуацию можно исправить, добавив еще одно ребро; используйте следующий код и проверьте снова:

```
AGraph.add_edge(1, 6)
sorted(nx.connected_components(AGraph))
```

На этот раз вы увидите желаемый результат:

```
[[1, 2, 3, 4, 5, 6]]
```

Теперь вывод показывает, что каждый узел соединен как минимум с одним другим узлом. Однако вы не знаете, у каких узлов наибольшее

количество соединений. Количество ребер, инцидентных конкретному узлу, называется *степенью*. Чем выше степень, тем сложнее становится узел. Зная степень, вы можете понять, какие узлы наиболее важны. Следующий код показывает, как получить степени для примера графа:

```
print([val for (node, val) in AGraph.degree()])
```

При запуске этого кода вы увидите степени для каждого узла по порядку:

```
[4, 2, 3, 2, 2, 1]
```

Значения степеней отображаются в порядке узлов: у узла 1 четыре соединения, а у узла 6 — только одно. Следовательно, узел 1 наиболее важный, за ним следует узел 3, имеющий три соединения.

При моделировании реальных данных, например твитов на определенную тему, у узлов также есть тенденция к кластеризации. Можно рассматривать эту тенденцию как своего рода тренд — то, что люди считают важным в данную минуту. Математический термин для такой тенденции — *кластеризация*, и измерение этой тенденции помогает понять, какая группа узлов наиболее важна в графе. Вот код, который используется для измерения кластеризации в примере графа:

```
nx.clustering(AGraph)
```

Результат этого измерения оказывается интересным:

```
{1: 0.16666666666666666, 2: 1.0, 3: 0.3333333333333333,
4: 0.0, 5: 0.0, 6: 0.0}
```

Вывод показывает, что узлы с наибольшей вероятностью группируются вокруг узла 2, хотя у узла 1 наивысшая степень. Это происходит потому, что у обоих узлов, 1 и 3, высокие степени, а узел 2 находится между ними.



ЗАПОМНИТЕ

Кластеризация графов помогает лучше понять данные. Этот метод позволяет выявить узлы с более сильными связями и узлы, находящиеся на грани изоляции. Понимание того, как элементы связаны в графе, дает возможность определить способы укрепления его структуры или, наоборот, ее разрушения. Во времена холодной войны военные ученые как из США, так и из стран советского блока изучали кластеризацию графов, чтобы лучше понять, как нарушить цепочки поставок противника в случае конфликта.

ИСПОЛЬЗОВАНИЕ ПРОБЕЛОВ В ВЫВОДЕ

В книге вывод для данного примера размещен на двух строках, хотя в Jupyter Notebook он отображается на одной строке. Добавление пробелов помогает сделать вывод более читаемым на странице – это не влияет на саму информацию. В других примерах книги вывод может быть представлен и в несколько строк, даже если в Jupyter Notebook он помещается в одну строку.

Вычисление центральности

Существует несколько видов центральности, поскольку важность часто зависит от различных факторов. Важные элементы графа при анализе твитов будут отличаться от важных элементов при анализе транспортных потоков. К счастью, NetworkX предоставляет ряд методов для расчета центральности. Например, можно вычислить центральность на основе степеней узлов. Следующий код использует модифицированный граф из предыдущего блока главы. (Этот код можно найти в файле **A4D2E; 08; Graph Centrality.ipynb** из загружаемых исходных кодов; подробнее о том, где найти этот файл, см. во введении.)

```
import networkx as nx
import matplotlib.pyplot as plt

AGraph = nx.Graph()

Nodes = range(1,6)
Edges = [(1,2), (2,3), (3,4), (4,5),
         (1,3), (1,5), (1,6)]

AGraph.add_nodes_from(Nodes)
AGraph.add_edges_from(Edges)

nx.degree_centrality(AGraph)
```

При запуске этого кода вы увидите следующий результат:

```
{1: 0.8, 2: 0.4, 3: 0.6000000000000001, 4: 0.4, 5: 0.4,
6: 0.2}
```

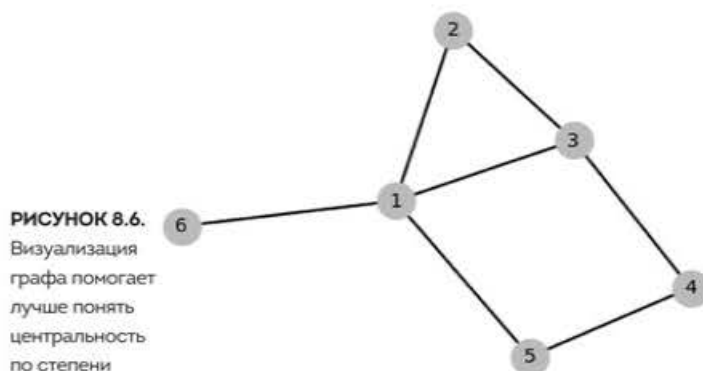
Значения различаются в зависимости от количества связей для каждого узла. Так как у узла 1 есть четыре соединения (наибольшая степень),

у него также наивысшая центральность. Вы можете увидеть, как это работает, построив граф с помощью следующего кода:

```
'with_labels': True,
'node_color': 'skyblue',
'node_size': 700, 'width': 2,
'font_size': 14,
'pos': nx.layout.spring_layout(AGraph, seed=5))

nx.draw(AGraph, **draw_params)
```

Этот код создает изображение, показанное на рисунке 8.6.



Узел 1 действительно находится в центре графа и имеет наибольшее количество соединений. Степень узла 1 делает его самым важным с точки зрения количества связей. При работе с ориентированными графами вы также можете использовать функции `in_degree_centrality()` и `out_degree_centrality()` для определения центральности по степени на основе типа связи, а не просто количества соединений.

При анализе дорожного движения может потребоваться определить, какие локации являются центральными на основе их расстояния до других узлов. Даже если у торгового центра в пригороде есть много связей, то, что он находится в пригороде, может уменьшить его влияние на трафик. В то же время супермаркет в центре города с небольшим количеством связей может сильно влиять на движение, поскольку находится близко ко многим другим узлам. Чтобы увидеть, как это работает, добавим еще один узел, 7, который не связан с графом. Центральность этого узла бесконечна, поскольку никакой другой узел не может до него добраться. Следующий код показывает, как вычислить центральность по близости для различных узлов в примере графа:

```
AGraph.add_node(7)
nx.closeness_centrality(AGraph)
```

При запуске этого кода вы увидите следующий результат:

```
{1: 0.6944444444444445,  
2: 0.5208333333333334,  
3: 0.5952380952380952,  
4: 0.462962962962963,  
5: 0.5208333333333334,  
6: 0.4166666666666667,  
7: 0.0}
```

Результат показывает центральность каждого узла в графе на основе его близости ко всем остальным узлам. Обратите внимание, что узел 7 имеет значение 0, что означает бесконечное расстояние до всех остальных узлов. С другой стороны, узел 1 имеет высокое значение, поскольку он близок ко всем узлам, с которыми имеет связь. Рассчитывая центральность по близости, вы можете определить, какие узлы наиболее важные с точки зрения их расположения.

Другая форма центральности по расстоянию — центральность по посредничеству. Представьте, что вы управляете компанией, которая перевозит товары по городу. Вам нужно знать, какие узлы сильнее всего влияют на эти перевозки. Возможно, вы сможете перенаправить часть трафика в обход такого узла, чтобы сделать работу эффективнее. При расчете центральности по посредничеству определяется узел, через который проходит наибольшее количество коротких путей. Вот код, используемый для этого расчета (все еще с присутствующим отключенным узлом 7):

```
nx.betweenness_centrality(AGraph)
```

Результат этой проверки:

```
{1: 0.36666666666666664,  
2: 0.0,  
3: 0.13333333333333333,  
4: 0.03333333333333333,  
5: 0.06666666666666667,  
6: 0.0,  
7: 0.0}
```

Как и ожидалось, узел 7 не влияет на передачу между другими узлами, поскольку не имеет с ними связей. Аналогично, поскольку узел 6 — это листовой узел с единственным соединением, он тоже не влияет на передачи. Взгляните снова на рисунок 8.6. Подграф, состоящий из узлов 1, 3, 4 и 5, сильнее всего влияет на передачу элементов в данном случае. Между узлами 1 и 4 нет прямого соединения, поэтому узлы 3 и 5 выступают в роли посредников. В этом случае узел 2 ведет себя как листовой узел.



СОВЕТ

NetworkX предоставляет ряд других функций для расчета центральности. Полный список этих функций можно найти по адресу <https://networkx.org/documentation/stable/reference/algorithms/centrality.html>. Важно определить, как именно вы хотите рассчитывать важность узлов. Крайне необходимо рассматривать центральность с учетом того типа важности, который вы хотите присвоить вершинам и ребрам графа.

§ 4. Перевод графа в числовой формат

Точность — важная часть использования алгоритмов. Хотя избыточная точность может скрыть общую картину от человека, компьютеры процветают на детализации. Часто чем больше деталей вы можете предоставить, тем лучшие результаты вы получите. Однако форма этих деталей имеет значение. Для использования определенных алгоритмов данные должны быть представлены в определенных форматах, иначе у полученного результата не будет смысла (он будет содержать ошибки или другие проблемы).

К счастью, NetworkX предоставляет ряд функций для преобразования вашего графа в форматы, которые могут использовать другие пакеты и среды. Эти функции можно найти по адресу <https://networkx.org/documentation/stable/reference/convert.html>. В следующих блоках показано, как представить данные графа в виде матрицы NumPy (<https://numpy.org>), разреженного представления SciPy (<https://scipy.org>) и стандартного списка Python. По мере изучения книги вы будете использовать эти представления для работы с различными алгоритмами. (Код в следующих блоках находится в файле `A4D2E; 08; Graph Conversion.ipynb` и опирается на граф, который вы создали ранее в этой главе в блоке «Подсчет ребер и вершин».)

Преобразование графа в матрицу

С помощью NetworkX можно легко преобразовать граф в матрицу NumPy и обратно для выполнения различных задач. NumPy используется для всевозможных операций с данными. Анализируя данные в графе, вы можете увидеть закономерности, которые обычно не заметны. Вот код, используемый для преобразования графа в матрицу, понятную для NumPy:


```
import networkx as nx
import matplotlib.pyplot as plt

AGraph = nx.Graph()
Nodes = range(1,6)
Edges = [(1,2), (2,3), (3,4), (4,5),
         (1,3), (1,5), (1,6)]

AGraph.add_nodes_from(Nodes)
AGraph.add_edges_from(Edges)

nx.to_numpy_matrix(AGraph)
```

Результат выполнения этого кода — следующая матрица:

```
matrix([[ 0.,  1.,  1.,  0.,  1.,  1.],
        [ 1.,  0.,  1.,  0.,  0.,  0.],
        [ 1.,  1.,  0.,  1.,  0.,  0.],
        [ 0.,  0.,  1.,  0.,  1.,  0.],
        [ 1.,  0.,  0.,  1.,  0.,  0.],
        [ 1.,  0.,  0.,  0.,  0.,  0.]])
```

Строки и столбцы результирующей матрицы показывают, где существуют связи. Например, между узлом 1 и ним самим нет связи, поэтому в строке 1, столбце 1 стоит 0. Однако между узлом 1 и узлом 2 есть связь, поэтому вы видите 1 в строке 1, столбце 2 и строке 2, столбце 1 (что означает, что связь идет в обоих направлениях как неориентированное соединение).

Размер этой матрицы зависит от количества узлов (у матрицы столько же строк и столбцов, сколько узлов), и когда она становится огромной, ей приходится представлять множество узлов, поскольку общее количество ячеек равно квадрату числа узлов. Например, невозможно представить интернет с помощью такой матрицы, потому что, по консервативным оценкам, при 1010 веб-сайтов потребуется матрица с 1020 ячейками для хранения его структуры, что невозможно при нынешних вычислительных мощностях.

Кроме того, количество узлов влияет на содержимое матрицы. Если n — это число узлов, то минимальное количество единиц будет равно 0, а максимальное — n^2 . То, сколько единиц в матрице — много или мало, определяет, является ли граф плотным или разреженным. Это важно, поскольку при небольшом количестве связей между узлами, как, например, в случае с веб-сайтами, существуют более эффективные способы хранения данных графа.

Использование разреженных представлений

Пакет SciPy тоже выполняет различные математические, научные и инженерные задачи. При использовании этого пакета можно применять разреженную матрицу для хранения данных. Разреженная матрица — это матрица, в которой хранятся только реально существующие связи, все остальные элементы отсутствуют. Использование разреженной матрицы экономит ресурсы, поскольку требования к памяти для такой матрицы невелики. Вот код, который используется для создания разреженной матрицы SciPy из графа NetworkX:

```
print(nx.to_scipy_sparse_matrix(AGraph))
```

Вот какой вывод вы можете ожидать от этого кода:

```
(0, 1) 1
(0, 2) 1
(0, 4) 1
(0, 5) 1
(1, 0) 1
(1, 2) 1
(2, 0) 1
(2, 1) 1
(2, 3) 1
(3, 2) 1
(3, 4) 1
(4, 0) 1
(4, 3) 1
(5, 0) 1
```

Как видите, записи показывают различные координаты ребер. С каждой активной координатой связана 1. Координаты начинаются с 0. Это означает, что (0, 1) фактически указывает на связь между узлами 1 и 2.

Использование списка для хранения графа

В зависимости от ваших потребностей вам может понадобиться и возможность создания словаря списков. Многие разработчики используют этот подход для создания кода, выполняющего различные задачи анализа графов. Один из таких примеров вы можете найти по адресу <https://www.geeksforgeeks.org/generate-graph-using-dictionary-python>. Следующий код показывает, как создать словарь списков для примера графа:

```
nx.to_dict_of_lists(AGraph)
```

Вот представление графа в виде словаря списков (обратите внимание, что порядок чисел может различаться):

```
{1: [2, 3, 5, 6], 2: [1, 3], 3: [1, 2, 4], 4: [3, 5],  
5: [1, 4], 6: [1]}
```

Заметьте, что каждый узел представляет собой элемент словаря; за ним следует список узлов, с которыми он соединен. Например, узел 1 соединен с узлами 2, 3, 5 и 6.

В ЭТОЙ ГЛАВЕ

- » Работа с графами
- » Выполнение задач сортировки
- » Уменьшение размера дерева
- » Поиск кратчайшего пути между двумя точками

ГЛАВА 9

Восстанавливая связи

Системы глобального позиционирования (GPS) работают благодаря тому, что можно использовать алгоритмы для обхода точек на карте и улиц, которые их соединяют. Фактически к концу этой главы вы поймете принципы работы GPS-навигатора (хотя и не обязательно механику его реализации). Конечно, фундаментальным требованием для использования графа при создании GPS-навигатора является возможность поиска связей между точками на карте, как описано в первом параграфе главы.

Чтобы разобраться в графе, нужно отсортировать узлы, как описано во втором параграфе главы, создав определенную организацию. Без организации принятие любого решения становится невозможным. Алгоритм может заикнуться или выдать неудобные результаты.

Когда вы смотрите на карту, вы не изучаете информацию в правом нижнем углу, если вам нужно работать с местоположениями и дорогами в верхнем левом углу. Компьютер не знает, что ему нужно смотреть в определенное место, пока вы ему это не укажете. Чтобы сфокусировать внимание на конкретной области, нужно уменьшить размер графа, как описано в третьем параграфе главы.

После упрощения задачи алгоритм может найти кратчайший маршрут между двумя точками, как описано в четвертом параграфе главы. В конце концов, вы не хотите проводить в пробках больше времени, чем необходимо, пробираясь от дома к месту назначения (и обратно). Концепция поиска кратчайшего маршрута немного сложнее, чем может показаться,

поэтому в четвертом параграфе подробно рассматриваются некоторые специфические требования к выполнению задач маршрутизации.



ЗАПОМНИТЕ

Вам не нужно вводить исходный код для этой главы вручную. На самом деле, использовать загружаемый исходный код намного проще. Вы можете найти исходный код для главы в файлах `A4D2E; 09; Graph Traversing.ipynb`, `A4D2E; 09; Minimum Spanning Tree.ipynb` и `A4D2E; 09; Shortest Path.ipynb` из загружаемых исходников. Подробнее о том, как найти эти исходные файлы, смотрите во введении.

§ 1. Эффективное прохождение графа

Обход графа означает поиск (посещение) каждой вершины (узла) в определенном порядке. Процесс посещения вершины может включать как чтение, так и обновление ее данных. При обходе графа непосещенная вершина считается *необнаруженной*. После посещения вершина становится *обнаруженной* (поскольку вы только что ее посетили), или *обработанной* (поскольку алгоритм проверил все исходящие из нее ребра). Порядок обхода определяет тип выполняемого поиска, и для этой задачи есть много алгоритмов. В следующих блоках рассматриваются два таких алгоритма.

УЧЕТ ИЗБЫТОЧНОСТИ

При обходе дерева каждый путь заканчивается листовым узлом, поэтому вы знаете, что достигли конца пути. Однако при работе с графом узлы взаимосвязаны таким образом, что для исследования всего графа может потребоваться обойти некоторые узлы более одного раза. По мере увеличения плотности графа возрастает вероятность многократного посещения одного и того же узла. Плотные графы могут значительно увеличить как вычислительные затраты, так и требования к памяти.

Чтобы уменьшить негативные последствия многократного посещения узла, обычно каждый посещенный узел помечается определенным образом, показывающим, что алгоритм уже в нем побывал. Когда алгоритм обнаруживает, что конкретный узел уже был посещен, он может просто пропустить его и перейти к следующему. Маркировка посещенных узлов снижает штрафы производительности, присущие избыточности.

Маркировка посещенных узлов позволяет также проверить завершенность поиска. В противном случае алгоритм может заикнуться и продолжать бесконечно обходить граф по кругу.

Создание графа

Чтобы увидеть, как может работать обход графа, вам нужен граф. Примеры в этом блоке основаны на общем графе, чтобы вы могли увидеть, как работают оба метода. Следующий код показывает список смежности, представленный в конце главы 8:

```
graph = {'A': ['B', 'C'],
        'B': ['A', 'C', 'D'],
        'C': ['A', 'B', 'D', 'E'],
        'D': ['B', 'C', 'E', 'F'],
        'E': ['C', 'D', 'F'],
        'F': ['D', 'E']}
```

Граф содержит двунаправленный путь, который проходит через вершины *A*, *B*, *D* и *F* с одной стороны (начиная с узла *A*) и *A*, *C*, *E* и *F* — с другой (снова начиная с узла *A*). Есть также соединения (которые действуют как возможные сокращения пути), идущие от *B* к *C*, от *C* к *D* и от *D* к *E*. Используя пакет NetworkX, представленный в главе 8, вы можете отобразить граф в виде рисунка, чтобы увидеть, как выглядят вершины и ребра (см. рисунок 9.1), используя следующий код:

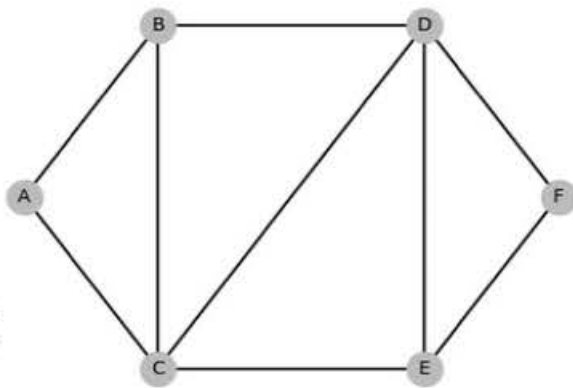
```
import networkx as nx
import matplotlib.pyplot as plt

Graph = nx.Graph()
for node in graph:
    Graph.add_nodes_from(node)
    for edge in graph[node]:
        Graph.add_edge(node, edge)

pos = {'A': [0.00, 0.50], 'B': [0.25, 0.75],
       'C': [0.25, 0.25], 'D': [0.75, 0.75],
       'E': [0.75, 0.25], 'F': [1.00, 0.50]}
draw_params = {'with_labels': True,
               'node_color': 'skyblue',
               'node_size': 700, 'width': 2,
               'font_size': 14}

nx.draw(Graph, pos, **draw_params)
plt.show()
```


РИСУНОК 9.1.
Представление
примера графа
с помощью
NetworkX



Применение поиска в ширину

Поиск в ширину (breadth-first search, BFS) начинается с начального узла и исследует каждый узел, присоединенный к корню. Затем исследует следующий уровень — изучая каждый уровень по очереди, пока не достигнет конца. Следовательно, в примере графа поиск исследует путь от *A* к *B* и *C* прежде, чем перейдет к исследованию *D*. Поиск в ширину исследует граф систематически, обходя вершины вокруг начальной вершины по кругу. Он начинает с посещения всех вершин на расстоянии одного шага от начальной вершины, затем переходит на два шага, затем на три шага и т.д. Следующий код демонстрирует, как выполнить поиск в ширину:

```
def bfs(graph, start):
    queue = [start]
    queued = [start]
    path = list()
    while queue:
        print('Queue is: %s' % queue)
        vertex = queue.pop(0)
        print('Processing %s' % vertex)
        for candidate in graph[vertex]:
            if candidate not in queued:
                queued.append(candidate)
                queue.append(candidate)
                path.append(vertex + '>' + candidate)
                print('Adding %s to the queue'
                      % candidate)
    return path
```

```
steps = bfs(graph, 'A')
print('\nBFS:', steps)
steps = bfs(graph, 'A')
print('\nBFS:', steps)
```

Вот вывод, который печатается при выполнении фрагмента кода:

```
Queue is: ['A']
Processing A
Adding B to the queue
Adding C to the queue
Queue is: ['B', 'C']
Processing B
Adding D to the queue
Queue is: ['C', 'D']
Processing C
Adding E to the queue
Queue is: ['D', 'E']
Processing D
Adding F to the queue
Queue is: ['E', 'F']
Processing E
Queue is: ['F']
Processing F

BFS: ['A>B', 'A>C', 'B>D', 'C>E', 'D>F']
```



ЗАПОМНИТЕ

Вывод показывает, как работает алгоритм поиска. Он идет в ожидаемом порядке — один уровень за раз. Главное преимущество использования BFS заключается в том, что он гарантированно возвращает кратчайший путь между двумя точками в качестве первого результата при использовании для поиска путей.



СОВЕТ

В примере кода для реализации очереди используется простой список. Как описано в блоке «Использование очередей» главы 6, очередь — это структура данных типа «первым пришел — первым ушел» (FIFO), которая работает как очередь в банке, где первый добавленный элемент также первым извлекается. Python предоставляет еще более удобную структуру данных под названием *deque* (произносится как «дек»), которую можно использовать и как очередь, и как стек. Подробнее о функции *deque* можно узнать на странице <https://pymotw.com/2/collections/deque.html>.

Применение поиска в глубину

Помимо поиска в ширину (BFS), для обхода вершин графа можно использовать поиск в глубину (depth-first search, DFS). При выполнении DFS алгоритм начинает с начальной вершины и исследует каждую вершину, двигаясь по одному пути до конца. Затем возвращается назад и начинает исследовать непройденные пути в текущем маршруте поиска, пока снова не достигнет начальной вершины. И если из начальной вершины доступны другие пути, алгоритм выбирает один из них и начинает тот же процесс поиска заново. Идея заключается в том, чтобы полностью исследовать каждый путь, прежде чем переходить к следующему.

Чтобы этот метод поиска работал, алгоритм должен помечать каждую посещенную вершину. Таким образом он знает, какие вершины требуют посещения, и может определить, какой путь выбрать следующим. Выбор между BFS и DFS может быть значим в зависимости от того, как вам нужно обходить граф. С точки зрения программирования разница между этими двумя алгоритмами заключается в том, как они хранят вершины для исследования:

- » **очередь для BFS** – список, работающий по принципу FIFO (недавно обнаруженные вершины недолго ждут обработки);
- » **стек для DFS** – список, работающий по принципу «последним пришел – первым ушел» (LIFO).

Следующий код показывает, как реализовать DFS:

```
def dfs(graph, start):
    stack = [start]
    parents = {start: start}
    path = list()
    while stack:
        print('Stack is: %s' % stack)
        vertex = stack.pop(-1)
        print('Processing %s' % vertex)
        for candidate in graph[vertex]:
            if candidate not in parents:
                parents[candidate] = vertex
                stack.append(candidate)
                print('Adding %s to the stack'
                      % candidate)
        path.append(parents[vertex] + '>' + vertex)
    return path[1:]
```



```
steps = dfs(graph, 'A')
print('\nDFS:', steps)
```

Вот результат выполнения этого фрагмента кода:

```
Stack is: ['A']
Processing A
Adding B to the stack
Adding C to the stack
Stack is: ['B', 'C']
Processing C
Adding D to the stack
Adding E to the stack
Stack is: ['B', 'D', 'E']
Processing E
Adding F to the stack
Stack is: ['B', 'D', 'F']
Processing F
Stack is: ['B', 'D']
Processing D
Stack is: ['B']
Processing B

DFS: ['A>C', 'C>E', 'E>F', 'C>D', 'A>B']
```

Последняя строка вывода показывает фактический порядок поиска. Обратите внимание, что поиск начинается с начальной вершины, как и ожидалось, но затем следует вниз по левой стороне графа до начала. Последний шаг — это поиск единственной ветви, отходящей от цикла, которая в данном случае создает граф, то есть вершину *D*.

Заметьте, что результат отличается от результата BFS. Маршрут обработки начинается с вершины *A* и перемещается на противоположную сторону графа, к вершине *F*. Затем код возвращается назад в поисках пропущенных путей. Такое поведение обусловлено использованием стека вместо очереди. Опора на стек означает, что вы также можете реализовать этот тип поиска с помощью рекурсии. Использование рекурсии сделает алгоритм быстрее, поэтому вы сможете получать результаты быстрее, чем при использовании поиска в ширину. Платой за это является то, что при использовании рекурсии расходуется больше памяти.

Выбор подходящего метода

Выбор между поиском в ширину и поиском в глубину зависит от того, как вы планируете использовать результаты поиска. Разработчики часто применяют поиск в ширину для максимально быстрого нахождения кратчайшего пути между двумя точками. Поэтому поиск в ширину часто используется в таких приложениях, как GPS, где нахождение кратчайшего маршрута имеет первостепенное значение. В рамках книги вы также увидите применение поиска в ширину для построения остовного дерева, поиска кратчайшего пути и многих других алгоритмов минимизации.

Поиск в глубину находит полный путь, прежде чем исследовать какой-либо другой маршрут. DFS используют, когда нужен детальный, а не общий поиск; он часто применяется в играх, где важно найти полный путь. Это также оптимальный подход для выполнения таких задач, как поиск выхода из лабиринта.



ЗАПОМНИТЕ

Выбор между поиском в ширину и поиском в глубину следует делать с учетом ограничений каждого метода. Поиск в ширину требует много памяти, поскольку систематически сохраняет все пути перед нахождением решения. Поиск в глубину требует меньше памяти, но нет гарантии, что он найдет кратчайшее и самое прямое решение.

§ 2. Сортировка элементов графа

Способность эффективно искать в графах зависит от сортировки. В конце концов, представьте, что вы пришли в библиотеку, где книги расставлены на полках в произвольном порядке. Поиск одной книги занял бы часы. Библиотека работает потому, что отдельные книги находятся в определенных местах, что делает их легко находимыми.

Библиотеки также демонстрируют еще одно свойство, которое важно при работе с определенными типами графов. При поиске книги вы начинаете с определенной категории, затем переходите к ряду книг, затем к полке в этом ряду и, наконец, к самой книге. При выполнении поиска вы двигаетесь от менее конкретного к более конкретному, что означает, что вы не возвращаетесь к предыдущим уровням. Таким образом, вы не оказываетесь в странных частях библиотеки, у которых нет ничего общего с интересующей вас темой.

В следующих блоках рассматриваются *направленные ациклические графы* (DAG), которые представляют собой конечные ориентированные графы без циклов. Другими словами, вы начинаете с определенного места

и следуете по конкретному маршруту к конечной точке, никогда не возвращаясь к начальной точке. У такого типа графа много практических применений, например для представления графиков работ, где каждая вершина представляет собой определенную веху.

ГРАФЫ С ЦИКЛАМИ

Иногда нужно описать процесс так, чтобы определенный набор шагов повторялся.

Например, когда вы моете машину, то ополаскиваете ее, наносите мыло, а затем снова ополаскиваете. Однако, если вы обнаруживаете грязное место – участок, который мыло не отчистило с первого раза, вы снова наносите мыло и ополаскиваете это место, чтобы убедиться, что пятно исчезло. Именно это и делает цикл: он создает ситуацию, в которой набор шагов повторяется одним из двух способов:

- **выполнено определенное условие** (например, пятно на машине исчезло);
- **выполнено заданное количество раз** (это максимальное число повторений, которое вы делаете, чтобы не повредить лакокрасочное покрытие автомобиля).

Работа с направленными ациклическими графами (DAG)

Направленные ациклические графы (DAG) — один из важнейших видов графов благодаря их широкому практическому применению. Основные принципы DAG заключаются в том, что они:

- » следуют определенному порядку, при котором невозможно попасть из одной вершины в другую и вернуться к начальной вершине, используя произвольный маршрут;
- » обеспечивают конкретный путь от одной вершины к другой (даже при наличии нескольких путей), позволяя создавать предсказуемые маршруты.

DAG используются для многих организационных задач. Например, генеалогическое древо — это пример DAG. Даже когда действия не следуют хронологическому или какому-либо другому преобладающему порядку, DAG позволяют создавать предсказуемые маршруты, что делает их проще в обработке в сравнении со многими другими типами графов.

Тем не менее у DAG могут быть необязательные маршруты. Представьте, что вы собираете бургер. Система меню начинается с нижней булочки. Вы можете по желанию добавить приправы на нижнюю булочку или сразу перейти к котлете. Маршрут всегда заканчивается котлетой, но существует несколько путей к ней. После того как котлета уложена, вы можете выбрать добавление сыра или бекона перед тем, как положить верхнюю булочку. Вы следуете определенному пути, но каждый путь может соединяться со следующим уровнем несколькими способами.

Использование топологической сортировки

Важным элементом DAG является то, что с их помощью можно представить множество различных действий. Однако некоторые действия требуют выполнения задач в определенном порядке. Здесь на помощь приходит *топологическая сортировка*. Она упорядочивает все вершины графа на прямой так, что прямые ребра указывают слева направо. При такой организации код может легко обходить граф и обрабатывать вершины одну за другой по порядку.

При использовании топологической сортировки граф организуется таким образом, что каждая вершина ведет к последующей вершине в последовательности. Например, при составлении графика строительства небоскреба вы не начинаете с верха и не спускаетесь. Вы начинаете с фундамента и поднимаетесь. Каждый этаж может представлять собой контрольную точку. Закончив второй этаж, вы не переходите к третьему, чтобы затем переделать второй. Вместо этого вы продвигаетесь от третьего этажа к четвертому и т.д. Любой вид планирования, требующий перемещения от конкретной начальной точки к конкретной конечной точке, может опираться на DAG с топологической сортировкой.

Топологическая сортировка может помочь определить, что в вашем графе нет циклов (поскольку в противном случае невозможно упорядочить ребра, соединяющие вершины, слева направо; как минимум одна вершина будет ссылаться на предыдущую). Кроме того, топологическая сортировка оказывается полезной в алгоритмах обработки сложных графов, так как показывает оптимальный порядок их обработки.

Топологическую сортировку можно получить с помощью алгоритма поиска в глубину (DFS). Достаточно просто отметить порядок обработки вершин алгоритмом. В предыдущем примере вывод происходит в следующем порядке: *A, C, E, F, D* и *B*. Если проследить последовательность на рисунке 9.1, можно заметить, что топологическая сортировка следует за ребрами по внешнему периметру графа. Затем совершается полный обход: после достижения последнего узла топологической сортировки остается всего один шаг до *A* — начала последовательности.

§ 3. Сведение к минимальному остовному дереву

Многие задачи, решаемые алгоритмами, основаны на определении минимума необходимых ресурсов, например на поиске экономичного способа достижения всех точек на карте. Эта проблема была особенно актуальна в конце XIX — начале XX века, когда во многих странах начали появляться железнодорожные и электрические сети, революционизировавшие транспорт и образ жизни. Строительство таких сетей частными компаниями было дорогостоящим (требовало много времени и рабочей силы). Использование меньшего количества материалов и рабочей силы позволяло экономить за счет сокращения избыточных соединений.



ЗАПОМНИТЕ

Некоторая избыточность желательна в критически важных транспортных или энергетических сетях, даже при стремлении к экономичным решениям. Если сеть соединена только одним способом, ее легко нарушить, что приведет к прерыванию обслуживания многих клиентов.

Исторический контекст поиска минимального остовного дерева

В Моравии, восточной части Чешской Республики, чешский математик Отакар Боровка нашел решение в 1926 году, позволяющее построить электрическую сеть с использованием минимально возможного количества проводов. Его решение весьма эффективно, поскольку не только позволило найти способ соединить все города Моравии наиболее экономичным способом, но и имело временную сложность $O(m \cdot \log n)$, где m — количество ребер (электрических кабелей), а n — количество вершин (городов). С тех пор решение Боровки было усовершенствовано другими. Хотя алгоритмы, которые вы найдете в книгах, лучше спроектированы и легче для понимания (такие, как алгоритмы Прима и Краскала), они не достигают лучших результатов с точки зрения временной сложности.

Минимальное остовное дерево определяет задачу поиска наиболее экономичного способа выполнения задачи. *Остовное дерево* — это список ребер, необходимых для соединения всех вершин в неориентированном графе. Один граф может содержать несколько остовных деревьев, в зависимости от расположения графа, и определение количества деревьев является сложной задачей. Каждый путь, который можно пройти от начала до завершения в графе, — это другое остовное дерево. Остовное дерево посещает каждую вершину только один раз; оно не закичивается и не повторяет элементы пути.

Работа с невзвешенными и взвешенными графами

Когда вы работаете с невзвешенным графом, у всех остовных деревьев одинаковая длина. В невзвешенных графах все ребра равной длины, и порядок их обхода не имеет значения, поскольку путь всегда одинаков. Все возможные остовные деревья содержат одинаковое количество ребер, $n - 1$ (где n — число вершин), одинаковой длины. Более того, любой алгоритм обхода графа, будь то поиск в ширину или поиск в глубину, подходит для нахождения одного из возможных остовных деревьев.

Ситуация усложняется при работе со взвешенным графом, где у ребер различные длины. В этом случае среди множества возможных остовных деревьев лишь у нескольких (или даже одного) будет минимально возможная длина. *Минимальное остовное дерево* — это такое остовное дерево, которое гарантирует путь с наименьшим возможным суммарным весом ребер. В неориентированном графе обычно существует только одно минимальное остовное дерево, хотя это зависит от конфигурации. Представьте минимальные остовные деревья так: глядя на карту, вы видите несколько путей из точки A в точку B . На каждом пути есть места, где нужно повернуть или сменить дорогу, — это вершины. Расстояние между вершинами представляет вес ребра. Как правило, один из путей между точками A и B оказывается кратчайшим.

Создание примера минимального остовного дерева

При построении минимальных остовных деревьев не всегда следует полагаться на очевидное. Например, при работе с картами вас может интересовать не расстояние, а время в пути, расход топлива или множество других параметров. Для каждого из этих параметров может получиться разное минимальное остовное дерево. Имея это в виду, на следующем примере вы лучше поймете концепцию минимальных остовных деревьев и увидите, как решить задачу нахождения наименьшего веса ребер для любой заданной проблемы. Чтобы продемонстрировать решение с использованием минимального остовного дерева на Python, следующий код обновляет предыдущий граф, добавляя веса ребер.

```
import networkx as nx
import matplotlib.pyplot as plt

graph = {'A': {'B':2, 'C':3},
         'B': {'A':2, 'C':2, 'D':2},
```



```

'C': {'A':3, 'B':2, 'D':3, 'E':2},
'D': {'B':2, 'C':3, 'E':1, 'F':3},
'E': {'C':2, 'D':1, 'F':1},
'F': {'D':3, 'E':1}}

Graph = nx.Graph()
for node in graph:
    Graph.add_nodes_from(node)
    for edge, weight in graph[node].items():
        Graph.add_edge(node,edge, weight=weight)

pos = { 'A': [0.00, 0.50], 'B': [0.25, 0.75],
        'C': [0.25, 0.25], 'D': [0.75, 0.75],
        'E': [0.75, 0.25], 'F': [1.00, 0.50]}

draw_params = {'with_labels':True,
               'arrows': True,
               'node_color':'skyblue',
               'node_size':700, 'width':2,
               'font_size':14}

labels = nx.get_edge_attributes(Graph,'weight')
nx.draw(Graph, pos, **draw_params)
nx.draw_networkx_edge_labels(Graph, pos,
                             font_size=14,
                             edge_labels=labels)

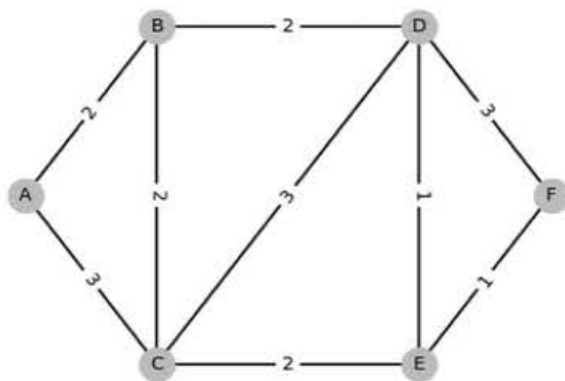
plt.show()

```

Рисунок 9.2 показывает, что теперь все ребра имеют значения. Эти значения могут представлять собой что угодно: время, топливо или деньги. Взвешенные графы могут отображать множество задач оптимизации, возникающих в географическом пространстве (например, перемещение между городами), поскольку представляют ситуации, когда можно входить в вершину и выходить из нее.

Примечательно, что в этом примере у всех ребер положительные веса. Однако у взвешенных графов могут быть отрицательные веса на некоторых ребрах. Отрицательные ребра находят применение во многих ситуациях. Например, они полезны, когда при перемещении между вершинами можно как получить выгоду, так и понести потери, скажем — при получении или потере денег во время транспортировки или торговли товарами либо при высвобождении энергии в химическом процессе.

РИСУНОК 9.2.
Пример графа
становится
взвешенным



ЗАПОМНИ

Не все алгоритмы хорошо справляются с обработкой отрицательных ребер. Важно отметить те из них, которые могут работать только с положительными весами.

Выбор правильных алгоритмов

Есть много различных алгоритмов для построения минимального остовного дерева. Наиболее распространены жадные алгоритмы, которые выполняются за полиномиальное время. *Полиномиальное время* — это степень от числа ребер, например $O(n^2)$ или $O(n^3)$ (дополнительную информацию о полиномиальном времени см. в блоке «Рассмотрение NP-полных задач» главы 15). Основные факторы, влияющие на скорость выполнения таких алгоритмов, связаны с процессом принятия решений, то есть с определением того, должно ли конкретное ребро входить в минимальное остовное дерево или превышает ли минимальный суммарный вес результирующего дерева определенное значение. С учетом этого вот некоторые из доступных алгоритмов для построения минимального остовного дерева:

- » **алгоритм Борувки.** Изобретенный Отакаром Борувкой в 1926 году, алгоритм основан на последовательности этапов, на каждом из которых определяются ребра с наименьшим весом. Вычисления начинаются с рассмотрения отдельных вершин, поиска наименьшего веса для каждой вершины и продолжается объединением путей для формирования лесов из отдельных деревьев, пока не будет создан путь, объединяющий все леса с наименьшим весом;
- » **алгоритм Прима.** Первоначально изобретенный Ярником в 1930 году и заново открытый Примом в 1957 году, алгоритм начинает с произвольной вершины и наращивает

минимальное остовное дерево по одному ребру за раз, всегда выбирая ребро с наименьшим весом, которое соединяет еще не включенную в дерево вершину с растущим деревом;

- » **алгоритм Краскала.** Разработанный Джозефом Краскалом в 1956 году, алгоритм использует подход, объединяющий алгоритм Борушки (создание лесов из отдельных деревьев) и алгоритм Прима (поиск минимального ребра для каждой вершины и построение лесов по одному ребру за раз).



СОВЕТ

Эти алгоритмы используют жадный подход. Жадные алгоритмы рассматриваются в главе 2 среди семейств алгоритмов, а подробно — описываются в главе 15. При *жадном подходе* алгоритм постепенно приходит к решению, принимая на каждом шаге наилучшее из доступных решений, причем это решение необратимое. Например, если вам нужен кратчайший путь между несколькими вершинами, жадный алгоритм выбирает кратчайшие ребра среди всех доступных между вершинами.

Знакомство с приоритетными очередями

Позже в главе вы увидите, как реализовать алгоритмы Прима и Краскала для построения минимального остовного дерева, а также алгоритм Дейкстры для поиска кратчайшего пути в графе с использованием Python. Однако, прежде чем это сделать, вам нужен метод для поиска ребер с минимальным весом среди множества ребер. Такая операция подразумевает упорядочивание, а упорядочивание элементов требует времени. Это сложная операция, как описано в главе 7. Поскольку в примерах постоянно происходит переупорядочивание ребер, здесь пригодится структура данных, называемая *приоритетной очередью*.

Используемые здесь *приоритетные очереди* основаны на древовидных структурах данных типа куч, которые позволяют быстро упорядочивать элементы при их вставке в кучу. Подобно волшебной шляпе фокусника, приоритетные кучи хранят ребра с их весами и мгновенно готовы предоставить вставленное ребро, вес которого минимален среди всех хранящихся.

В этом примере используется класс, который позволяет выполнять в приоритетной очереди сравнения, определяющие, содержит ли очередь элементы и когда эти элементы содержат определенное ребро (избегая двойных вставок). Приоритетная очередь обладает еще одной полезной характеристикой (польза которой объясняется при работе с алгоритмом Дейкстры): если вы вставляете ребро с весом, отличным от ранее сохраненного, код обновляет вес ребра и перестраивает позицию ребра в куче.


```

from heapq import heapify, heappop, heappush

class priority_queue():
    def __init__(self):
        self.queue = list()
        heapify(self.queue)
        self.index = dict()
    def push(self, priority, label):
        if label in self.index:
            self.queue = [(w,l)
                           for w,l in self.queue if l != label]
            heapify(self.queue)
        heappush(self.queue, (priority, label))
        self.index[label] = priority
    def pop(self):
        if self.queue:
            return heappop(self.queue)
    def __contains__(self, label):
        return label in self.index
    def __len__(self):
        return len(self.queue)

```

Использование алгоритма Прима

Алгоритм Прима создает минимальное остовное дерево для графа, обходя граф вершина за вершиной. Начиная с любой выбранной вершины, алгоритм добавляет ребра, используя ограничение, согласно которому, если одна вершина в настоящее время является частью остовного дерева, а вторая — нет, вес ребра между ними должен быть наименьшим возможным среди доступных. Если действовать таким образом, вероятность создания циклов в остовном дереве меньше (это могло бы произойти, если бы добавить ребро, вершины которого уже находятся в обоих остовных деревьях) и вы гарантированно получите минимальное дерево, поскольку добавляете ребра с наименьшим весом. С точки зрения этапов алгоритм включает следующие три фазы, последняя из которых итеративная:

- 1** отслеживание как ребер минимального остовного дерева, так и использованных вершин по мере их включения в решение;
- 2** начало с любой вершины в графе и помещение ее в решение;
- 3** определение, остались ли вершины, не входящие в решение:

а) перечисление ребер, которые соединяются с вершинами в текущем решении;

б) добавление в остовное дерево ребра с минимальным весом. (Здесь проявляется жадный принцип алгоритма: на каждом шаге выбирается минимальное значение, чтобы получить минимальный результат в целом.)

Переведя эти шаги в код на Python, вы можете протестировать алгоритм на примере взвешенного графа, используя следующий код:

```
def prim(graph, start):
    treepath = {}
    total = 0
    queue = priority_queue()
    queue.push(0, (start, start))
    while queue:
        weight, (node_start, node_end) = queue.pop()
        if node_end not in treepath:
            treepath[node_end] = node_start
            if weight:
                print("Added edge from %s \
                    " to %s weighting %i"
                    % (node_start, node_end, weight))
            total += weight
            for next_node, weight \
            in graph[node_end].items():
                queue.push(weight, (node_end, next_node))
    print("Total spanning tree length: %i" % total)
    return treepath

treepath = prim(graph, 'A')
treepath = prim(graph, 'A')
```

При выполнении код выводит следующий результат:

```
Added edge from A to B weighting 2
Added edge from B to C weighting 2
Added edge from B to D weighting 2
Added edge from D to E weighting 1
Added edge from E to F weighting 1
Total spanning tree length: 8
```

Алгоритм печатает шаги обработки, показывая добавляемое на каждом этапе ребро и вес, который это ребро добавляет к общей сумме. В примере отображается общая сумма весов, а алгоритм возвращает словарь

Python, содержащий конечную вершину в качестве ключа и начальную вершину — в качестве значения для каждого ребра полученного остовного дерева. Другая функция, `represent_tree()`, преобразует пары «ключ — значение» словаря в кортежи, а затем сортирует каждый из получившихся кортежей для лучшей читаемости пути в дереве:

```
def represent_tree(treepath):
    progression = list()
    for node in treepath:
        if node != treepath[node]:
            progression.append((treepath[node], node))
    return sorted(progression, key=lambda x:x[0])

print(represent_tree(treepath))
print(represent_tree(treepath))
```

Эта версия вывода немного более удобочитаема:

```
[(('A', 'B'), ('B', 'C'), ('B', 'D'), ('D', 'E'), ('E', 'F'))]
```



ЗАПОМНИТЕ

Функция `represent_tree()` переупорядочивает вывод алгоритма Прима для лучшей читаемости. Однако алгоритм работает с неориентированным графом, что означает возможность прохождения по ребрам в обоих направлениях. Алгоритм учитывает это допущение, поскольку при добавлении в очередь с приоритетом для последующей обработки не выполняется проверка направленности ребер.

Тестирование алгоритма Краскала

Алгоритм Краскала, как и алгоритм Прима, использует жадную стратегию, но выбирает кратчайшие ребра из глобального пула, содержащего все ребра. Чтобы определить, является ли ребро подходящей частью решения, алгоритм опирается на процесс агрегации, в котором он собирает вершины вместе. Когда ребро соединяет вершины, уже находящиеся в решении, алгоритм отбрасывает его, чтобы избежать создания цикла. Алгоритм действует следующим образом:

- 1 поместить все ребра в кучу и отсортировать их так, чтобы кратчайшие ребра оказались наверху;
- 2 создать набор деревьев, каждое из которых содержит только одну вершину (так что количество деревьев равно количеству вершин), и объединять деревья в агрегат до тех пор, пока они не сойдутся в дерево минимальной длины, охватывающее все вершины;

3 повторять следующие операции, пока решение не будет содержать столько ребер, сколько вершин в графе:

- а) выбрать кратчайшее ребро из кучи;
- б) определить, находятся ли две вершины, соединенные этим ребром, в разных деревьях из множества связанных деревьев;
- в) если деревья различны — соединить их с помощью этого ребра (определяя объединение);
- г) если вершины находятся в одном дереве — отбросить ребро;
- д) повторять шаги с «а» по «д» для оставшихся ребер в куче.

Следующий пример показывает, как преобразовать эти шаги в код на Python:

```
def kruskal(graph):
    priority = priority_queue()
    print("Pushing all edges into the priority queue")
    treepath = list()
    connected = dict()
    for node in graph:
        connected[node] = [node]
        for dest, weight in graph[node].items():
            priority.push(weight, (node, dest))
    print("Totally %i edges" % len(priority))
    print("Connected components: %s"
          % connected.values())

    total = 0
    while len(treepath) < (len(graph)-1):
        (weight, (start, end)) = priority.pop()
        if end not in connected[start]:
            treepath.append((start, end))
            print("Summing %s and %s components:"
                  % (connected[start], connected[end]))
            print("\tadded edge from %s " \
                  "to %s weighting %i"
                  % (start, end, weight))
            total += weight
            connected[start] += connected[end][:]
            for element in connected[end]:
                connected[element] = connected[start]
    print("Total spanning tree length: %i" % total)
    return sorted(treepath, key=lambda x: x[0])

print('\nMinimum spanning tree: %s' % kruskal(graph))
```

При выполнении алгоритм Краскала выдает следующий результат:

```
Pushing all edges into the priority queue
Totally 9 edges
Connected components: dict_values([[ 'A' ], [ 'E' ], [ 'F' ],
                                     [ 'B' ], [ 'D' ], [ 'C' ]])

Summing [ 'E' ] and [ 'D' ] components:
    added edge from E to D weighting 1
Summing [ 'E', 'D' ] and [ 'F' ] components:
    added edge from E to F weighting 1
Summing [ 'A' ] and [ 'B' ] components:
    added edge from A to B weighting 2
Summing [ 'A', 'B' ] and [ 'C' ] components:
    added edge from B to C weighting 2
Summing [ 'A', 'B', 'C' ] and [ 'E', 'D', 'F' ] components:
    added edge from B to D weighting 2
Total spanning tree length: 8

Minimum spanning tree:
[( 'A', 'B' ), ( 'B', 'C' ), ( 'B', 'D' ), ( 'E', 'D' ), ( 'E', 'F' )]
```



СОВЕТ

Алгоритм Краскала предлагает решение, похожее на то, что предлагает алгоритм Прима. Однако для разных графов алгоритмы Прима и Краскала могут давать различные решения минимального остовного дерева, поскольку каждый алгоритм идет разными путями к своим выводам.

Определение наилучшего алгоритма

Оба алгоритма — и Прима, и Краскала — выдают одну связную компоненту, соединяя все вершины графа одной из кратчайших последовательностей ребер (минимальным остовным деревом). Суммируя веса ребер, алгоритмы определяют длину полученного остовного дерева. Поскольку оба алгоритма дают рабочее решение, при выборе лучшего варианта нужно опираться на время выполнения и тип взвешенного графа.

Что касается времени выполнения, оба алгоритма показывают схожие результаты с оценкой сложности по *Big-O* равной $O(E \cdot \log(V))$, где E — количество ребер, а V — количество вершин. Однако нужно учитывать, как они решают задачу, поскольку есть различия в среднем ожидаемом времени выполнения.

Алгоритм Прима пошагово строит единственное решение, добавляя ребра, тогда как алгоритм Краскала создает набор частичных решений и объединяет их. При создании своего решения алгоритм Прима

опирается на более сложные структуры данных, чем алгоритм Краскала, поскольку постоянно добавляет потенциальные ребра в качестве кандидатов и продолжает выбирать кратчайшее ребро для продвижения к решению. При работе с плотным графом алгоритм Прима предпочтительнее алгоритма Краскала, поскольку его очередь с приоритетом на основе куч выполняет всю работу по сортировке быстро и эффективно.



ЗАПОМНИТЕ

В примере для выбора кратчайших ребер используется очередь с приоритетом на основе бинарной кучи, но существуют и более быстрые структуры данных, такие как *куча Фибоначчи*, которая может обеспечить лучшую производительность при большом количестве ребер в куче. При использовании кучи Фибоначчи временная сложность алгоритма Прима может измениться до $O(E + V \cdot \log(V))$, что явно выгоднее при большом количестве ребер (компонент E теперь прибавляется, а не умножается) в сравнении с ранее указанным временем выполнения $O(E \cdot \log(V))$.

Алгоритму Краскала не особо нужна очередь с приоритетом (хотя в одном из примеров она используется), поскольку перечисление и сортировка ребер происходят только один раз в начале процесса. Основанный на более простых структурах данных, работающих с отсортированными ребрами, он идеальный кандидат для обычных разреженных графов с меньшим количеством ребер.

§ 4. Поиск кратчайшего маршрута

Кратчайший путь между двумя точками — это необязательно прямая линия, особенно когда прямой линии в вашем графе нет. Допустим, вам нужно проложить электрические линии в населенном пункте. Кратчайший маршрут предполагал бы прокладку линий между каждым местом напрямую, без учета того, куда эти линии идут. Однако в реальной жизни простое решение обычно невозможно. Вероятно, придется прокладывать кабели вдоль дорог, а не через частную собственность, что означает поиск маршрутов, которые максимально сокращают расстояние с учетом этих ограничений.

Определение понятия поиска кратчайшего пути

Есть много приложений для алгоритмов поиска кратчайшего пути. Идея заключается в том, чтобы найти путь, предлагающий наименьшее расстояние между точками А и В. Поиск кратчайшего пути полезен как для транспортировки (как добраться до пункта назначения с минимальным расходом топлива), так и для коммуникации (как направить информацию, чтобы она пришла быстрее). Неожиданные применения задачи о кратчайшем пути могут возникнуть и в обработке изображений (для выделения контуров изображений), в играх (как достичь определенных игровых целей, используя минимальное количество ходов) и во многих других областях, где проблему можно свести к неориентированному или ориентированному взвешенному графу.

Алгоритм Дейкстры нашел наибольшее применение в решении задачи о кратчайшем пути. Эдсгер Вибе Дейкстра, голландский ученый-информатик, разработал этот алгоритм для демонстрации вычислительной мощности компьютера ARMAC (<http://www-set.win.tue.nl/UnsungHeroes/machines/armac.html>) в 1959 году. Изначально алгоритм решал задачу поиска кратчайших расстояний между 64 городами в Нидерландах на основе простой карты в виде графа.



ЗАПОМНИТЕ

Существуют и другие алгоритмы для решения задачи поиска кратчайшего пути. Алгоритмы Беллмана — Форда и Флойда — Уоршелла сложнее, но могут работать с графами, содержащими отрицательные веса. (Отрицательные веса могут создавать проблемы для алгоритма Дейкстры.) Хотя это не сразу очевидно, у отрицательных весов есть осмысленное представление в реальном мире. Если вы измеряете потребление энергии при перемещении между точками в пространстве и используете электрический двигатель, учтите, что при движении под гору электромобиль вырабатывает энергию, поэтому его энергопотребление для этой дуги в графе будет отрицательным. Другие примеры отрицательных весов можно найти в социальных сетях, когда учитываются как дизлайки, так и лайки, или при отображении финансовых транзакций в виде графа.

Поскольку задача поиска кратчайшего пути включает работу с графами, которые одновременно взвешенные и ориентированные, пример графа требует еще одного обновления перед тем, как продолжить (результат показан на рисунке 9.3).

```
import networkx as nx
import matplotlib.pyplot as plt

graph = {'A': {'B':2, 'C':3},
         'B': {'C':2, 'D':2},
```

```

'C': {'D':3, 'E':2},
'D': {'F':3},
'E': {'D':1, 'F':1},
'F': {}

Graph = nx.DiGraph()
for node in graph:
    Graph.add_nodes_from(node)
    for edge, weight in graph[node].items():
        Graph.add_edge(node,edge, weight=weight)

pos = { 'A': [0.00, 0.50], 'B': [0.25, 0.75],
        'C': [0.25, 0.25], 'D': [0.75, 0.75],
        'E': [0.75, 0.25], 'F': [1.00, 0.50]}

draw_params = {'with_labels':True,
               'arrows': True,
               'node_color':'skyblue',
               'node_size':700, 'width':2,
               'font_size':14}

nx.draw(Graph, pos, **draw_params)
nx.draw_networkx_edge_labels(Graph, pos,
                             font_size=14,
                             edge_labels=labels)

plt.show()

```

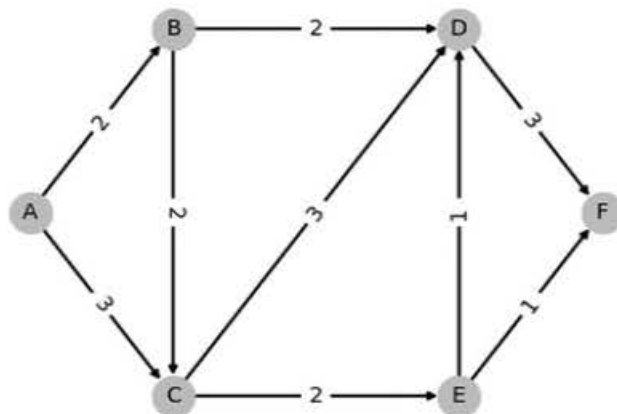


РИСУНОК 9.3.
Пример графа
становится
взвешенным
и ориентиро-
ванным

Добавление отрицательного ребра

В примере можно внести простые изменения в граф из предыдущего блока, чтобы вставить отрицательное ребро между узлами *B* и *C* (размещение отрицательных ребер показано на рисунке 9.4):

```
ngraph = {'A': {'B':2, 'C':3},
          'B': {'C':-1, 'D':2},
          'C': {'D':3, 'E':2},
          'D': {'F':3},
          'E': {'D':-1, 'F':1},
          'F': {}}

nGraph = nx.DiGraph()
for node in ngraph:
    nGraph.add_nodes_from(node)
    for edge, weight in ngraph[node].items():
        nGraph.add_edge(node, edge, weight=weight)
labels = nx.get_edge_attributes(nGraph, 'weight')

nx.draw(nGraph, pos, **draw_params)
nx.draw_networkx_edge_labels(nGraph, pos,
                             font_size=14,
                             edge_labels=labels)

plt.title("negative edges")
plt.show()
```

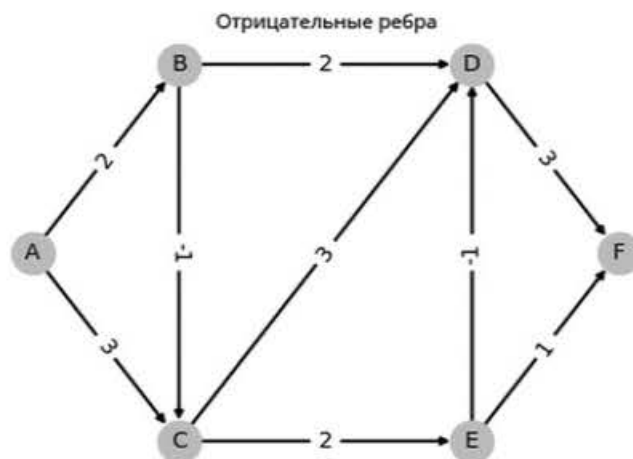


РИСУНОК 9.4.
К примеру графа добавлены отрицательные ребра

Особый случай возникает при наличии отрицательных циклов в графе. Наличие отрицательных циклов создает проблемы для всех алгоритмов вычисления кратчайшего пути. Необходимое условие для включения цикла с отрицательным весом в граф — использование отрицательных весов для ребер, но этого недостаточно. Нужно также иметь цикл, то есть последовательность циклически связанных узлов, сумма весов которых дает отрицательное число. Поскольку сумма цикла отрицательна, жадный алгоритм, пытающийся минимизировать общую стоимость проходимых ребер, может никогда не выйти из него и окажется в ловушке (как показано на рисунке 9.5 между A и C).

```
ncgraph = {'A': {'B':1},
          'B': {'C':1, 'D':2},
          'C': {'A':-3, 'D':3, 'E':2},
          'D': {'F':3},
          'E': {'D':1, 'F':1},
          'F': {}}

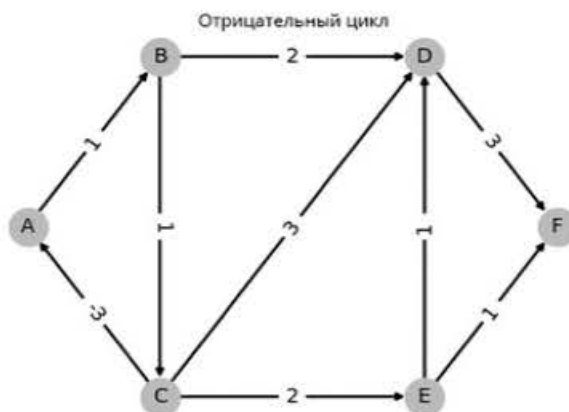
ncGraph = nx.DiGraph()
for node in ncgraph:
    ncGraph.add_nodes_from(node)
    for edge, weight in ncgraph[node].items():
        ncGraph.add_edge(node, edge, weight=weight)

labels = nx.get_edge_attributes(ncGraph, 'weight')

nx.draw(ncGraph, pos, **draw_params)
nx.draw_networkx_edge_labels(ncGraph, pos,
                             font_size=14,
                             edge_labels=labels)

plt.title("negative cycle")
plt.show()
```

РИСУНОК 9.5.
Отрицательный цикл в графе может создавать проблемы для некоторых алгоритмов



Эта модификация исходного графа создает отрицательный цикл вокруг узлов *A*, *B* и *C*: когда алгоритм достигает узла *C*, он не может вернуться к *A*, поскольку это менее затратный ход, и цикл может уменьшить общую стоимость пути.

Объяснение алгоритма Дейкстры

Алгоритм Дейкстры требует в качестве входных данных начальную и (опционально) конечную вершину. Если конечная вершина не указана, алгоритм вычисляет кратчайшее расстояние между начальной вершиной и всеми остальными вершинами графа. Когда конечная вершина определена, алгоритм останавливается при ее достижении и возвращает результат до этой точки, независимо от того, какая часть графа осталась неисследованной.

Алгоритм начинает с оценки расстояния от начальной точки до других вершин. Это исходное предположение записывается в очередь с приоритетом и по соглашению устанавливается равным бесконечности. Затем алгоритм приступает к исследованию соседних узлов, аналогично поиску в ширину (BFS). Это позволяет алгоритму определить, какие узлы находятся поблизости и что их расстояние равно весу соединяющих ребер. Эта информация сохраняется в очереди с приоритетом путем соответствующего обновления весов.



ЗАПОМНИТЕ

Естественно, алгоритм исследует соседей, поскольку они связаны с начальной вершиной направленным ребром. Алгоритм Дейкстры учитывает направление ребер.

На этом этапе алгоритм переходит к ближайшей вершине графа, основываясь на кратчайшем ребре в очереди с приоритетом. Технически алгоритм *посещает* новую вершину. Он начинает исследовать соседние вершины, исключая уже посещенные, определяет стоимость посещения каждой из непосещенных вершин и оценивает, меньше ли расстояние до них, чем расстояние, записанное в очереди с приоритетом.

Когда расстояние в очереди с приоритетом бесконечно, это означает, что алгоритм впервые посещает эту вершину и он записывает более короткое расстояние. Когда расстояние, записанное в очереди с приоритетом, небесконечно, но больше, чем только что вычисленное алгоритмом, это означает, что алгоритм нашел *короткий путь* — более короткий способ достичь этой вершины из начальной точки и сохраняет эту информацию в очереди с приоритетом. Разумеется, если расстояние, записанное в очереди с приоритетом, короче только что оцененного алгоритмом, алгоритм отбрасывает информацию, поскольку новый маршрут длиннее. После

обновления всех расстояний до соседних вершин алгоритм определяет, достигнута ли конечная вершина. Если нет, он выбирает кратчайшее ребро, присутствующее в очереди с приоритетом, посещает его и начинает оценивать расстояние до новых соседних вершин.



ЗАПОМНИТЕ

Как объясняется в описании алгоритма, алгоритм Дейкстры ведет точный учет стоимости достижения каждой встреченной вершины и обновляет информацию, только когда находит более короткий путь. Вычислительная сложность алгоритма при использовании бинарной кучи в *Big-O*-нотации составляет $O((E + V) \cdot \log(V))$, где E — количество ребер, а V — количество узлов (вершин) в графе. При использовании кучи Фибоначчи сложность может быть снижена до $O(E + V \cdot \log(V))$. Далее показано, как реализовать алгоритм Дейкстры на Python:

```
def dijkstra(graph, start, end):
    inf = float('inf')
    known = set()
    priority = priority_queue()
    path = {start: start}

    for vertex in graph:
        if vertex == start:
            priority.push(0, vertex)
        else:
            priority.push(inf, vertex)

    last = start
    while last != end:
        (weight, actual_node) = priority.pop()
        if actual_node not in known:
            for next_node in graph[actual_node]:
                upto_actual = priority.index[actual_node]
                upto_next = priority.index[next_node]
                to_next = upto_actual + \
                    graph[actual_node][next_node]
                if to_next < upto_next:
                    priority.push(to_next, next_node)
                    print("Found shortcut from %s to %s"
                          % (actual_node, next_node))
                    print("\tTotal length up so far: %i"
                          % to_next)
                    path[next_node] = actual_node
            last = actual_node
            known.add(actual_node)

    return priority.index, path
```



```
dist, path = dijkstra(graph, 'A', 'F')
```

После выполнения кода вы получите следующий вывод:

```
Found shortcut from A to C
  Total length up so far: 3
Found shortcut from A to B
  Total length up so far: 2
Found shortcut from B to D
  Total length up so far: 4
Found shortcut from C to E
  Total length up so far: 5
Found shortcut from D to F
  Total length up so far: 7
Found shortcut from E to F
  Total length up so far: 6
```

Алгоритм возвращает несколько полезных фрагментов информации: кратчайший путь до пункта назначения и минимальные зарегистрированные расстояния для посещенных вершин. Чтобы визуализировать кратчайший путь, вам нужна функция `reverse_path()`, которая перестраивает путь, делая его читаемым:

```
def reverse_path(path, start, end):
    progression = [end]
    while progression[-1] != start:
        progression.append(path[progression[-1]])
    return progression[::-1]

print(reverse_path(path, 'A', 'F'))
```

Вот результат, который вы увидите после выполнения кода:

```
['A', 'C', 'E', 'F']
```

Вы также можете узнать кратчайшее расстояние до каждого встреченного узла, запросив словарь `dist`:

```
print(dist)
```

Вывод показывает соответствие между узлом и расстоянием, как показано здесь:

```
{ 'D': 4, 'A': 0, 'B': 2, 'F': 6, 'C': 3, 'E': 5 }
```

Объяснение алгоритма Беллмана – Форда

Алгоритм Беллмана — Форда, вероятно, один из наиболее изучаемых алгоритмов на курсах информатики в университетах по всему миру, поскольку у него много важных практических применений. Этот алгоритм используется для поиска кратчайшего пути до каждого маршрутизатора в серверной сети, и он особенно хорошо для этого подходит, поскольку, в отличие от других алгоритмов, может быть распределен, то есть вычисления, необходимые для завершения работы алгоритма, могут распределяться между несколькими компьютерами, делая задачу быстрее и проще.

Алгоритм Беллмана — Форда, разработанный для решения той же задачи, что и алгоритм Дейкстры, использует другой подход к решению проблемы, что делает его более подходящим для мира сетевых компьютеров. Хотя алгоритм Дейкстры использует жадный подход, исследуя только ближайшие ребра перед принятием решения о конкретном пути, алгоритм Беллмана — Форда вместо этого проверяет и обрабатывает их все.

Этот выбор влияет на вычислительную сложность алгоритма в нотации *Big-O*, которая теперь составляет $O(E \cdot V)$, где E — количество ребер, а V — количество узлов (вершин) в графе. Однако такой выбор упрощает и разделение вычислений алгоритма на отдельные задачи, поскольку каждый компьютер может исследовать различный набор последовательных ребер.

Более эффективная маршрутизация информации

Реализация алгоритма Беллмана — Форда в этом блоке требует в качестве входных данных граф и начальную точку, поскольку рассматривает все остальные узлы графа как конечные точки. Начиная с исходной точки, он отслеживает два вида информации: расстояние до каждого конечного узла (которое изначально установлено как бесконечное) и последний узел, который заставил алгоритм обновить информацию о расстоянии до каждого другого узла. Оба вида информации записываются с помощью словаря Python.

Затем алгоритм перебирает каждый узел, а после — каждое ребро графа (основываясь на исследуемом узле). Это исчерпывающий поиск по структуре ребер графа. На каждом шаге итерации алгоритм вычисляет фактическое расстояние от начального узла до оцениваемого узла, добавляя исследуемое ребро. Если сумма этих двух величин представляет более короткий путь, чем известный в настоящее время, информация обновляется как в словаре расстояний, так и в словаре последних узлов (поскольку этот словарь представляет предыдущий узел в более коротком пути).

```
def bellman_ford(graph, start):
    inf = float('inf')
    distance = {node: inf if node!=start else 0.0 for node in graph}
    previous = {node: None for node in graph}

    for actual_node in graph:
        for next_node in graph[actual_node]:
            edge_weight = graph[actual_node][next_node]
            tempDistance = (distance[actual_node] +
                            edge_weight)
            if tempDistance < distance[next_node]:
                distance[next_node] = tempDistance
                previous[next_node] = actual_node
    return distance, previous
```

Когда все итерации завершены, функция возвращает словари расстояний и предыдущих узлов для дальнейших вычислений.

Доказательство универсальности алгоритма

Алгоритм Беллмана — Форда универсальнее, чем другие алгоритмы. Даже если вы не можете сразу использовать возвращаемые им расстояния, дальнейшие манипуляции с данными предоставляют более полезную информацию. Во-первых, повторно перебирая ребра графа, можно обнаружить несоответствия. Сложение кратчайшего расстояния до узла со стоимостью ребра для достижения другого целевого узла и сравнение его с кратчайшим расстоянием до этого целевого узла дают подсказку о наличии отрицательного цикла при обнаружении несоответствия, поскольку сумма всегда должна быть больше или равна кратчайшему расстоянию. При наличии отрицательного цикла оценки расстояний с использованием алгоритма Беллмана — Форда не являются точными. Во-вторых, используя информацию о предыдущем узле, вы можете восстановить кратчайший путь от исходного узла до любого другого узла.

```
def detect_negative_cycle(graph, distance, previous):
    for actual_node in graph:
        for next_node in graph[actual_node]:
            edge_weight = graph[actual_node][next_node]
            if distance[actual_node] + edge_weight <
                distance[next_node]:
                return True
    return False

def retrace_shortest_path(previous, end):
```



```

path = [end]
while path[0] != 'A':
    path = [previous[end]] + path
    end = previous[end]
return path

```

Используя следующий код, вы можете увидеть, как две предыдущие функции — `detect_negative_cycle()` и `retrace_shortest_path()` — работают с исходными данными графа:

```

distance, previous = bellman_ford(graph, 'A')
print(distance)
print("path:", retrace_shortest_path(previous, 'F'))
neg_cycle = detect_negative_cycle(graph, distance, previous)
print(f"There are negative cycles in the graph: {neg_cycle}")

```

Результат возвращает точные расстояния до каждого целевого узла и путь от узла A до узла F. Что важнее: вы знаете, что в графе нет отрицательных циклов, и можете доверять результатам.

```

{'A': 0.0, 'B': 2.0, 'C': 3.0, 'D': 4.0, 'E': 5.0, 'F': 6.0}
path: ['A', 'C', 'E', 'F']
There are negative cycles in the graph: False

```

Следующий код тестирует алгоритм на графе, содержащем ребро с отрицательным весом:

```

distance, previous = bellman_ford(ngraph, 'A')
print(distance)
print("path:", retrace_shortest_path(previous, 'F'))
neg_cycle = detect_negative_cycle(ngraph, distance, previous)
print(f"There are negative cycles in the graph: {neg_cycle}")

```

Полученный результат указывает на новый кратчайший путь от узла A до узла F, на этот раз — с использованием ребра с отрицательным весом. Отсутствие отрицательных циклов гарантирует корректность вычислений.

```

{'A': 0.0, 'B': 2.0, 'C': 1.0, 'D': 2.0, 'E': 3.0, 'F': 4.0}
path: ['A', 'B', 'C', 'E', 'F']
There are negative cycles in the graph: False

```

Наконец, вы можете протестировать алгоритм на графе, содержащем отрицательный цикл:

```

distance, previous = bellman_ford(ncgraph, 'A')
print(distance)
print("path:", retrace_shortest_path(previous, 'F'))

```

```
neg_cycle = detect_negative_cycle(ncgraph, distance, previous)
print(f"There are negative cycles in the graph: {neg_cycle}")
```

Вывод похож на предыдущие результаты, но с двумя существенными различиями: одно из кратчайших расстояний отрицательное (достижение начального узла из начального узла) и функция обнаружения отрицательных циклов подняла тревогу.

```
{'A': -1.0, 'B': 1.0, 'C': 2.0, 'D': 3.0, 'E': 4.0, 'F': 5.0}
path: ['A', 'B', 'C', 'E', 'F']
There are negative cycles in the graph: True
```

В итоге нельзя доверять результатам, полученным для этого графа, поскольку он содержит отрицательный цикл. Просматривая несколько узлов и ребер, можно легко заметить ребро с отрицательным весом, но на более крупных графах для быстрой проверки можно полагаться только на алгоритм Беллмана — Форда.

Объяснение алгоритма Флойда — Уоршелла

Роберт Флойд, специалист по информатике и лауреат Премии Тьюринга 1978 года, разработал алгоритм Флойда — Уоршелла в его нынешнем виде. Существуют и другие формы алгоритма, сформулированные другими учеными:

- » Бернаром Руа в 1959 году.
- » Стивеном Уоршеллом в 1962 году.

Он также похож на алгоритм Клини, опубликованный в 1956 году.

Алгоритм Флойда — Уоршелла является алгоритмом динамического программирования, поскольку решает перекрывающиеся подзадачи (что позволяет избежать повторных вычислений) и его решение получается с использованием решений его подзадач (свойство, называемое *оптимальной подструктурой*).

Алгоритм Флойда — Уоршелла, как и другие алгоритмы, рассмотренные в книге, может вычислять расстояния между узлами в графе и корректно работает с ребрами, имеющими отрицательный вес. Он выводит квадратную матрицу, связывающую все узлы графа с их относительными расстояниями (технически называемую матрицей расстояний). Поскольку алгоритм выполняет три вложенных прохода по узлам графа, его временная сложность зависит исключительно от количества узлов и составляет $O(V^3)$, где V обозначает вершины, то есть узлы.

Сравнение с другими алгоритмами

Хотя производительность алгоритма Флойда — Уоршелла похожа на производительность алгоритмов Дейкстры и Беллмана — Форда, есть несколько важных различий:

- » Алгоритм Дейкстры работает лучше всего (имеет меньшую сложность) при поиске расстояния между двумя узлами в графе, хотя не может корректно работать, если у некоторых ребер в графе отрицательный вес.
- » Алгоритм Беллмана — Форда вычисляет расстояние от одной вершины до всех остальных вершин графа. Он корректно работает даже при отрицательных весах ребер, но выполняется медленнее алгоритма Дейкстры.
- » Алгоритм Флойда — Уоршелла находит расстояния между всеми парами вершин графа и делает это достаточно эффективно, хотя он самый медленный из трех алгоритмов, поскольку выдает гораздо больше информации. Впрочем, в некоторых случаях алгоритмы Дейкстры и Беллмана — Форда могут дать тот же результат, что и алгоритм Флойда — Уоршелла, но потребуют больше времени и вычислений.

И алгоритм Беллмана — Форда, и алгоритм Флойда — Уоршелла могут работать с отрицательными весами при отсутствии отрицательных циклов. В свою очередь, алгоритм Дейкстры при наличии отрицательных весов ребер использует эту ситуацию, чтобы избежать посещения всех вершин. В таком случае результаты, полученные алгоритмом Дейкстры, будут недостоверными.

Иная ситуация возникает при наличии в графе отрицательных циклов. В этом случае алгоритм Беллмана — Форда завершится неудачей и не вернет решение, тогда как алгоритм Флойда — Уоршелла все равно завершит выполнение. Более того, отрицательный цикл можно обнаружить, если в результирующей матрице у какой-либо вершины есть отрицательное расстояние до самой себя (расстояния до самой себя находятся на диагонали матрицы).

Преимущество использования трех проходов

Данная реализация алгоритма Флойда — Уоршелла использует две функции. Первая из них, для заданного графа, определяет расстояние между двумя смежными вершинами; в противном случае функция возвращает нулевое расстояние, если это одна и та же вершина, или бесконечное расстояние, если вершины различны.


```
def dist(graph, start, end):
    if end in graph[start]:
        return float(graph[start][end])
    elif start == end:
        return 0.0
    else:
        return float('inf')
```

Используя функцию `dist()`, легко создать начальную матрицу расстояний для алгоритма Флойда — Уоршелла в виде словаря словарей, способа представления квадратной матрицы расстояний, с помощью которой можно определить кратчайшее расстояние между начальной вершиной (представленной строками) и конечной вершиной (представленной столбцами). Основная идея алгоритма заключается в том, чтобы, начав с квадратной матрицы, содержащей только известные расстояния между смежными вершинами, расширить ее до матрицы, содержащей кратчайшие пути между вершинами (если они достижимы; в противном случае расстояние остается бесконечным).

```
def floyd_warshall(graph):

    mat = {row: {col: dist(graph, row, col)
                  for col in graph} for row in graph}
    for k in mat:
        for i in mat:
            for j in mat:
                if mat[i][j] > mat[i][k] + mat[k][j]:
                    mat[i][j] = mat[i][k] + mat[k][j]
    return mat
```

Алгоритм работает с использованием трех вложенных циклов. Во всех циклах происходит перебор вершин. В ходе итераций алгоритм проверяет расстояния между опорной вершиной (из первой итерации, *k* в коде), начальной вершиной (из второй итерации, *i* в коде) и конечной вершиной (из третьей итерации, *j* в коде). Поскольку это триангуляция, алгоритм обновляет расстояние между начальной и конечной вершиной, если оно больше, чем расстояние при достижении конечной вершины из начальной через опорную вершину. Пройдя через все этапы алгоритма, вы можете быть уверены, что алгоритм вычислит точную матрицу расстояний, если в графе нет отрицательных циклов.

Функция возвращает матрицу расстояний в качестве результата. Поскольку результат представлен в виде словаря словарей, нужен фрагмент кода, чтобы преобразовать его в список списков — структуру данных, содержащую метки строк и столбцов для удобства чтения. Конечно, можно использовать и другие структуры данных, например `DataFrame`

из библиотеки Pandas (<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>), где метки строк и столбцов можно представить лучше, но в этом примере используется более простой подход для наглядности:

```
def print_mat(mat):
    els = [item for item in mat]
    labels = [' '] + [item for item in els]
    printable = [labels]
    for k, row in enumerate(els):
        printable.append([labels[k+1]] + [mat[row][col] for col in els])
    return printable
```

Чтобы определить наличие отрицательных циклов, нужно рассмотреть, что в таких обстоятельствах происходит с узлами, начальное расстояние которых равно нулю. Обычно такая ситуация возникает при вычислении расстояния от узла до самого себя. В этом случае, поскольку алгоритм Флойда — Уоршелла ищет альтернативные, более короткие пути через другой узел, вы никогда не найдете альтернативный узел, если только исходный узел не часть отрицательного цикла. Следовательно, если при наблюдении за диагональю матрицы расстояний, полученной алгоритмом, где лежат пути расстояний до одних и тех же узлов, вы обнаружите какое-либо отрицательное значение, это указывает на наличие отрицательных циклов в графе. Следующий код просто извлекает диагональ матрицы расстояний:

```
def extract_diagonal(mat):
    return [mat[item][item] for item in mat]
```

Теперь, когда код завершен, вы можете запустить алгоритм на примере графа:

```
print_mat(floyd_warshall(graph))
```

Вы видите следующую матрицу расстояний и читаете ее, находя начальный узел в метках строк (в крайнем левом столбце) и конечный узел — в метках столбцов (в первой строке). Диагональ, представляющая расстояние от узла до самого себя, должна быть заполнена нулевыми значениями. Если в ячейке матрицы вы находите бесконечное значение (*inf*), это означает, что в графе нет способа достичь этого узла из данной начальной точки (это обычно происходит в ориентированных графах).

```
[[' ', 'A', 'B', 'C', 'D', 'E', 'F'],
 ['A', 0.0, 2.0, 3.0, 4.0, 5.0, 6.0],
 ['B', inf, 0.0, 2.0, 2.0, 4.0, 5.0],
 ['C', inf, inf, 0.0, 3.0, 2.0, 3.0],
```

```
['D', inf, inf, inf, 0.0, inf, 3.0],  
['E', inf, inf, inf, 1.0, 0.0, 1.0],  
['F', inf, inf, inf, inf, inf, 0.0]]
```

Вы также можете проверить матрицу расстояний графа с отрицательным ребром:

```
print_mat(floyd_warshall(ngraph))
```

Результат показывает, что у некоторых путей отрицательные значения:

```
[[' ', 'A', 'B', 'C', 'D', 'E', 'F'],  
['A', 0.0, 2.0, 1.0, 2.0, 3.0, 4.0],  
['B', inf, 0.0, -1.0, 0.0, 1.0, 2.0],  
['C', inf, inf, 0.0, 1.0, 2.0, 3.0],  
['D', inf, inf, inf, 0.0, inf, 3.0],  
['E', inf, inf, inf, -1.0, 0.0, 1.0],  
['F', inf, inf, inf, inf, inf, 0.0]]
```

Наконец, вы можете построить матрицу расстояний для графа с отрицательным циклом и извлечь его диагональ:

```
mat = floyd_warshall(ncgraph)  
extract_diagonal(mat)
```

Полученная диагональ содержит отрицательные значения:

```
[-1.0, -1.0, -2.0, 0.0, 0.0, 0.0]
```



ЗАПОМНИТЕ

Не существует кратчайших путей между любой парой ребер, являющихся частью отрицательного цикла. Это происходит потому, что длина, создаваемая любыми такими путями, будет произвольно отрицательной в зависимости от проходов алгоритма.

- » Рассмотрение социальных сетей в виде графов
- » Взаимодействие с содержимым графа

ГЛАВА 10

Раскрывая секреты графов

Глава 8 помогает понять основы графов с точки зрения математики. Глава 9 расширяет ваши знания, помогая увидеть связь графов с алгоритмами. Эта глава фокусируется на практическом применении теорий из предыдущих двух глав для работы с графами.

Первый параграф описывает характер социальных сетей с помощью графов. В нем рассматривается важность связей, создаваемых социальными сетями. Например, анализ диалогов может выявить закономерности, которые помогут лучше понять основную тему, чем просто чтение этих диалогов. Определенная ветвь обсуждения может привлекать больше внимания, потому что она важнее другой. Конечно, такой анализ нужно проводить с учетом таких проблем, как спам. Подобный анализ может привести к интересным выводам, например о том, где лучше потратить рекламный бюджет, чтобы привлечь максимум внимания и, следовательно, продаж.

Второй параграф посвящен навигации по графам для достижения конкретных результатов. Например, при вождении вам может понадобиться узнать лучший маршрут между двумя точками с учетом того, что, хотя один маршрут короче, на нем ведутся строительные работы, из-за чего второй маршрут становится предпочтительнее. Иногда для поиска оптимального маршрута или наилучшего решения нужно использовать элемент случайности. Этот вопрос также обсуждается в данном параграфе.



ЗАПОМНИТЕ

Вам не нужно вводить исходный код для этой главы вручную. На самом деле, использовать загружаемый исходный код намного проще. Исходный код главы можно найти в файле `A4D2E; 10; Graph Secrets.ipynb` из загружаемых материалов. Подробнее о том, как найти этот файл, смотрите во введении.

§ 1. Представление социальных сетей как графов

Каждое социальное взаимодействие неизбежно связано со всеми другими социальными взаимодействиями того же типа. Например, возьмем любую социальную сеть. Ссылки на вашей странице соединяют вас с членами семьи, но также ведут к внешним источникам, которые, в свою очередь, связаны с другими внешними источниками. У каждого члена вашей семьи тоже есть внешние ссылки. Прямые и косвенные связи между различными страницами в итоге соединяют все страницы между собой, хотя для перехода с одной страницы на другую может потребоваться много промежуточных звеньев. Связность проявляется и во многих других формах. Суть в том, что изучать социальные сети, просто просматривая страницу или другой источник информации, сложно. *Анализ социальных сетей* (SNA) — это процесс изучения взаимодействий в социальных сетях с помощью графов, называемых *социограммами*, где узлы (например, страница) отображаются как точки, а связи (например, ссылки на внешние страницы) — как линии. В следующих блоках обсуждаются некоторые вопросы, связанные с изучением социальных сетей как графов.

Кластеризация сетей на группы

Люди склонны формировать сообщества — кластеры других людей со схожими идеями и взглядами. При изучении этих кластеров становится проще приписывать определенное поведение группе в целом (хотя приписывать общее поведение отдельному человеку и опасно, и ненадежно). Идея, лежащая в основе изучения кластеров, заключается в том, что если между людьми существует связь, то у них часто есть общий набор идей и целей. Находя кластеры, вы можете определять эти идеи, анализируя состав группы. Например, часто пытаются найти кластеры людей при выявлении страхового мошенничества и при налоговых проверках. Неожиданные группы людей могут вызвать подозрение в том, что они часть группы мошенников или уклонистов от уплаты налогов, поскольку отсутствуют обычные причины для объединения людей в таких обстоятельствах.

Графы дружбы могут представлять то, как люди связаны друг с другом. Узлы представляют отдельных людей, а ребра — их связи, такие как семейные отношения, деловые контакты или дружеские узы. Обычно графы дружбы неориентированные, поскольку представляют взаимные отношения, и иногда взвешены для отображения силы связи между двумя людьми.

При поиске кластеров в графе дружбы связи между узлами в этих кластерах зависят от триад, по сути — особых видов треугольников. Связи между тремя людьми могут попадать в следующие категории:

- » **замкнутые** — все трое знают друг друга (подумайте о семейной обстановке, где каждый знает всех остальных);
- » **открытые** — один человек знает двух других людей, но эти двое не знают друг друга (представьте человека, который знает одного человека на работе и другого — дома, но человек с работы ничего не знает о человеке из дома);
- » **связанная пара** — один человек знает одного из других людей в триаде, но не знает третьего человека (эта ситуация включает двух людей, которые что-то знают друг о друге и встречаются кого-то нового — того, кто потенциально хочет стать частью группы);
- » **несвязанные** — триада образует группу, но никто в группе не знает друг друга. (Последний вариант может показаться немного странным, но подумайте о конференции или семинаре. Люди на этих мероприятиях образуют группу, но могут ничего не знать друг о друге. Однако, поскольку у них схожие интересы, вы можете использовать кластеризацию для понимания поведения группы.)



СОВЕТ

Многие исследования фокусируются на неориентированных графах, которые отражают только связи между объектами. Однако можно также использовать ориентированные графы, чтобы показать, что человек *A* знает о существовании человека *B*, но человек *B* даже не подозревает о существовании человека *A*. В этом случае нужно рассматривать 16 различных типов триад. Для простоты в этой главе мы сосредоточимся только на четырех типах: замкнутых, открытых, связанных парах и несвязанных.

Триады естественным образом возникают в отношениях, и многие социальные сети в интернете использовали эту идею для ускорения установления связей между участниками. Плотность связей важна для любой социальной сети, поскольку связанная сеть может эффективнее распространять информацию и делиться контентом.

Пример в этом блоке основан на графе карате-клуба Захари, описанном на странице https://documentation.sas.com/doc/en/pgmsascdc/v_009/casmlnetwork/casmlnetwork_network_examples06.htm. Это небольшой граф, который позволяет понять, как работают сети, без необходимости тратить много времени на загрузку большого набора данных. К счастью, этот набор данных входит в состав пакета *networkx*, представленного в главе 8. Сеть карате-клуба Захари отображает дружеские отношения

между 34 членами карате-клуба с 1970 по 1972 год. Социолог Уэйн У. Захари использовал его в качестве предмета исследования. Он написал об этом статью под названием «Модель информационного потока для конфликтов и расколов в малых группах». Интересный факт об этом графе и статье заключается в том, что в те годы в клубе возник конфликт между одним из инструкторов по карате (узел номер 0) и президентом клуба (узел номер 33). С помощью кластеризации графа можно почти идеально предсказать последующий раскол клуба на две группы.

Поскольку этот пример строит граф, показывающий группы (чтобы их было легче визуализировать), вам также понадобится пакет `matplotlib`. Следующий код демонстрирует, как отобразить узлы и ребра набора данных. (Вы можете найти этот код в файле исходного кода **A4D2E; 10; Social Networks.ipynb**; подробнее см. во введении.)

```
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

graph = nx.karate_club_graph()

pos=nx.spring_layout(graph)
nx.draw(graph, pos, with_labels=True)
plt.show()
```

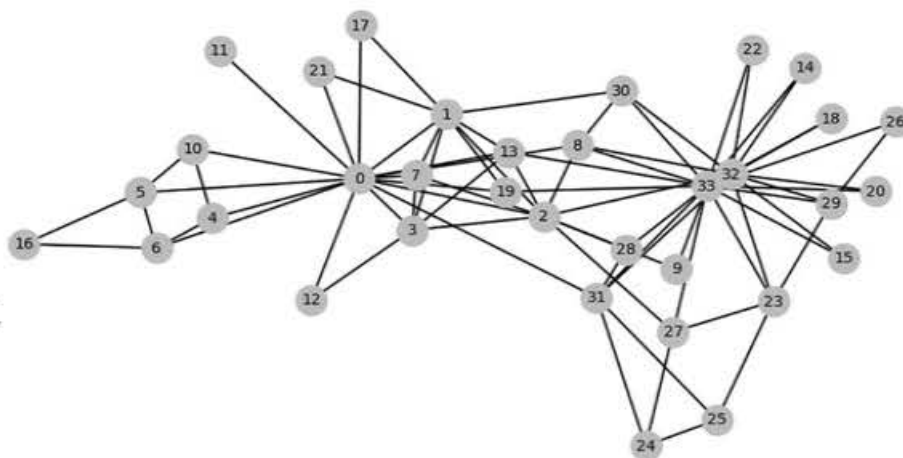
Чтобы отобразить график на экране, вам также нужно задать компоновку, определяющую, как размещать узлы на экране. В этом примере используется силовой алгоритм Фрюхтермана — Рейнгольда (вызов `nx.spring_layout`). Однако вы можете выбрать один из других макетов, описанных в разделе «Graph Layout» по адресу <https://networkx.github.io/documentation/networkx-1.9/reference/drawing.html>. На рисунке 10.1 показан результат работы примера. (Ваш результат может выглядеть иначе из-за особенностей генерации графа алгоритмом.)



ЗАПОМНИТЕ

Силовой алгоритм Фрюхтермана — Рейнгольда для автоматической компоновки графов создает понятные макеты с разделенными узлами и ребрами, которые стараются не пересекаться, имитируя физическое взаимодействие электрически заряженных частиц или магнитов одного знака. Глядя на выходной граф, можно заметить, что у некоторых узлов только одно соединение, у некоторых — два, а у некоторых — более двух. Ребра образуют триады, как упоминалось ранее. Однако самое важное — то, что рисунок 10.1 наглядно показывает кластеризацию, возникающую в социальной сети.

РИСУНОК 10.1.
Граф, показыва-
ющий сетевые
кластеры вза-
имоотношений
друзей



Обнаружение сообществ

Группа тесно связанных людей часто определяет сообщество. Фактически термин *клика* применяется к группе, членство в которой эксклюзивно и где все хорошо знают друг друга. У большинства людей есть детские воспоминания о тесной группе друзей в школе или в районе, которые всегда проводили время вместе. Это и есть клика, в которой каждый член связан со всеми остальными.



ЗАПОМНИТЕ

Клики больше ассоциируются с неориентированными графами, чем с ориентированными. В ориентированных графах особое внимание уделяется связным компонентам, где между всеми парами узлов в компоненте существует путь прямого соединения. Город — пример сильно связного компонента, поскольку вы можете добраться до любого пункта назначения из любой начальной точки, следуя по одно- и двусторонним улицам.

С математической точки зрения клика — это еще более строгое понятие, поскольку подразумевает *подграф* (часть сетевого графа, которую можно выделить как самостоятельный целостный элемент), обладающий максимальной связностью. При анализе различных социальных сетей легко выделить кластеры, но сложнее найти клики — группы с максимальной связностью — внутри этих кластеров. Зная расположение клик, можно лучше понять сплоченность сообщества. Кроме того, эксклюзивный характер клик часто приводит к формированию группы с собственными правилами, отличными от правил социальной сети в целом. Следующий пример показывает, как извлечь клики и сообщества из графа каратеклуба, использованного в предыдущем блоке (обратите внимание, что оператор импорта в первых двух строках должен быть записан в одну строку):


```

from networkx.algorithms.community.kclique import
k_clique_communities

graph = nx.karate_club_graph()
# Finding and printing all cliques of four
cliques = nx.find_cliques(graph)
print ('All cliques of four: %s'
        % [c for c in cliques if len(c)>=4])

# Joining cliques of four into communities
communities = nx.k_clique_communities(graph, k=4)
communities_list = [list(c) for c in communities]
nodes_list = [node for community in communities_list for
               node in community]
print ('Found these communities: %s' % communities_list)

# Printing the subgraph of communities
subgraph = graph.subgraph(nodes_list)
nx.draw(subgraph, with_labels=True)
plt.show()

All cliques of four: [[0, 1, 2, 3, 13], [0, 1, 2, 3, 7],
                     [33, 32, 8, 30], [33, 32, 23, 29]]
Found these communities: [[0, 1, 2, 3, 7, 13],
                          [32, 33, 29, 23], [32, 33, 8, 30]]

```

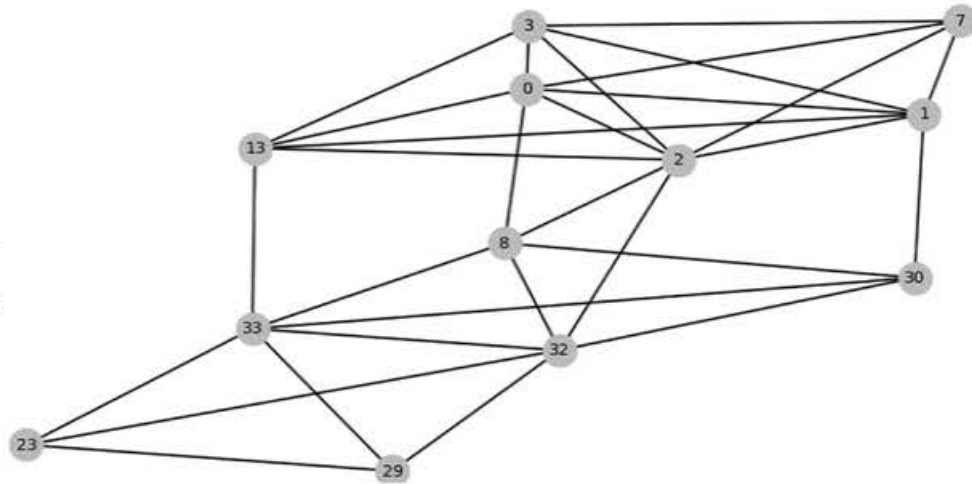
Пример начинается с извлечения узлов из набора данных карате-клуба, имеющих четыре или более соединения, а затем выводит клики с размером не менее четырех. Конечно, вы можете установить любой уровень связности между узлами, который считаете релевантным. Возможно, вы рассматриваете клику как сообщество, в котором у каждого узла есть 20 соединений, в то время как другие могут считать кликой сообщество, где у каждого узла всего три соединения.

В завершение вы можете нарисовать подграф и отобразить его. На рисунке 10.2 показан результат выполнения этого примера, который отображает набор клик с четырьмя или более соединениями.

Поиск клик в графах — сложная задача, требующая множества вычислений. (Она трудная, поскольку количество клик может экспоненциально расти с добавлением каждого нового узла.) Задачу можно решить с помощью алгоритма, основанного на полном переборе, что означает проверку всех возможных подмножеств узлов, чтобы определить, являются ли они кликами. Однако можно использовать и более умный

подход, например — алгоритм, разработанный голландскими учеными Конрадом Броном и Джоном Кербошем в 1973 году, который находит все максимальные клики в заданном графе.

РИСУНОК 10.2.
Сообщества
часто содержат
клики, которые
могут быть
полезны для
анализа соци-
альных сетей



Алгоритм работает на основе рекурсии и систематического исследования. В итоге он выдает список всех максимальных клик. Клика считается максимальной, когда к ней нельзя добавить больше узлов. Для работы алгоритм использует три множества вершин, традиционно обозначаемых как R — для потенциальных клик, P — для оставшихся неисследованных узлов и X — для узлов, которые нужно пропустить, поскольку они уже были исследованы (алгоритм не повторяется). Вначале R и X пусты, а P содержит все узлы графа. Алгоритм работает путем перебора узлов из множества P , рассматривая каждый в качестве кандидата в клику. От итерации к итерации алгоритм рассматривает только соседей кандидата, помещая их в R , а остальные перемещает во множество X . По завершении исследования результирующая максимальная клика возвращается из множества R , содержащего все узлы, являющиеся соседями друг для друга. Хотя алгоритм одновременно рекурсивный и итеративный, на практике он считается одним из самых быстрых алгоритмов поиска клик, поскольку в худшем случае время его выполнения для графа с n вершинами составляет $O(3n/3)$.

Однако один только список клик не очень полезен, если хотите увидеть сообщества. Для этого нужно использовать специализированные и сложные алгоритмы, чтобы объединять пересекающиеся клики и искать кластеры, такие как метод перколяции клик, описанный на <https://www.salatino.org/wp/cliq-percolation-method-in-python>. Пакет NetworkX предлагает `k_clique_communities` — реализацию алгоритма перколяции клик, который объединяет все клики определенного размера (параметр k). Эти клики определенного размера имеют $k - 1$ общих элементов (то есть отличаются всего одним компонентом; это действительно строгое правило).

Алгоритм перколяции клик предоставляет список всех найденных сообществ. В данном примере одна клика формируется вокруг инструктора по карате, а другая — вокруг президента клуба. Кроме того, вы можете объединить все узлы, входящие в сообщество, в единый набор, что помогает создать подграф, состоящий только из сообществ.

§ 2. Навигация графа

Навигация, или *обход*, графа означает посещение каждого узла графа. Целью навигации по графу может быть определение содержимого узлов или его обновление по мере необходимости. При обходе графа вполне возможно, что код посетит определенные узлы более одного раза из-за связности, которую обеспечивают графы. Следовательно, нужно также рассмотреть возможность пометки узлов как посещенных после просмотра их содержимого. Процесс навигации по графу важен для определения способов соединения узлов, чтобы выполнять различные задачи. В предыдущих главах обсуждались базовые методы навигации по графу. Следующие блоки помогут вам понять некоторые более продвинутые методы навигации по графу.

Подсчет степеней разделения

Термин *степени разделения* определяет расстояние между узлами в графе. При работе с неориентированным графом без взвешенных ребер каждое ребро считается за одну степень разделения. Однако при работе с другими типами графов, например с картами, где каждое ребро может представлять значение расстояния или времени, степени разделения могут существенно различаться. Суть в том, что степени разделения указывают на некоторое расстояние. Пример в этом блоке (и следующий за ним) основан на следующих данных графа (этот код можно найти в загружаемом файле исходного кода `A4D2E; 10; Graph Navigation.ipynb`; подробнее см. во введении):

```
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

data = {'A': ['B', 'F', 'H'],
        'B': ['A', 'C'],
        'C': ['B', 'D']}
```

```

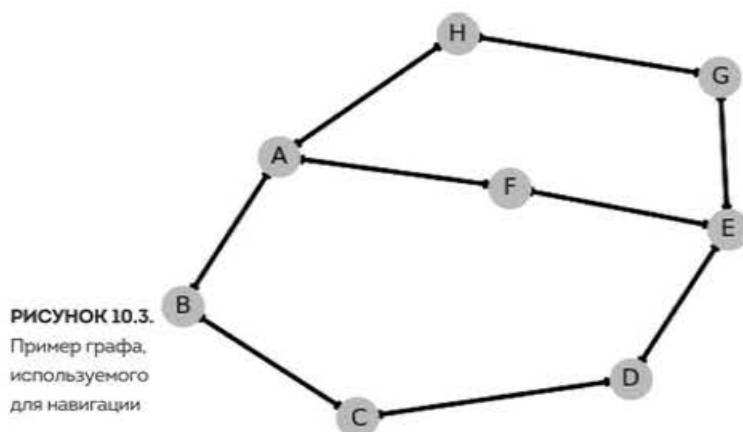
'D': ['C', 'E'],
'E': ['D', 'F', 'G'],
'F': ['E', 'A'],
'G': ['E', 'H'],
'H': ['G', 'A']]

graph = nx.DiGraph(data)
pos=nx.spring_layout(graph)
nx.draw_networkx_labels(graph, pos, font_size=16)
nx.draw_networkx_nodes(graph, pos, node_shape="o",
                        node_color='skyblue', node_size=800)
nx.draw_networkx_edges(graph, pos, width=1, arrows=True,
                        arrowstyle='->', arrowsize=20)

plt.show()

```

Это расширенная версия графа, использованного в блоке «Выходим за рамки деревьев» главы 6 (рисунок 6.2). На рисунке 10.3 показано, как выглядит этот граф, чтобы вы могли визуализировать, что делает вызов функции. Обратите внимание, что это ориентированный граф (*networkx DiGraph*), поскольку у использования ориентированного графа есть определенные преимущества при определении степеней разделения (и выполнении многих других вычислений).



Чтобы определить степени разделения между двумя элементами, необходима начальная точка. В данном примере можно использовать узел A. Следующий код показывает необходимый вызов функции пакета *networkx* и его вывод:

```

nx.shortest_path_length(graph, 'A')

{'A': 0, 'B': 1, 'C': 2, 'D': 3, 'E': 2, 'F': 1, 'G': 2,
 'H': 1}

```


Расстояние между узлом A и узлом A, конечно же, равно 0. Наибольшая степень разделения наблюдается между узлом A и узлом D, которая равна 3. Такую информацию можно использовать для определения маршрута или анализа соотношения затрат на топливо и времени для различных путей. Суть в том, что знание кратчайшего расстояния между двумя точками может быть крайне важным. Пакет `networkx`, использованный в этом примере, содержит широкий набор алгоритмов измерения расстояний, описанных на странице https://networkx.org/documentation/stable/reference/algorithms/shortest_paths.html.



ПРИМЕЧАНИЕ

Чтобы увидеть, как использование ориентированного графа может существенно повлиять на расчеты степеней разделения, попробуйте удалить связь между узлами A и F. Измените данные следующим образом:

```
data = {'A': ['B', 'H'],
        'B': ['A', 'C'],
        'C': ['B', 'D'],
        'D': ['C', 'E'],
        'E': ['D', 'F', 'G'],
        'F': ['E', 'A'],
        'G': ['E', 'H'],
        'H': ['G', 'A']}
```

Когда вы выполните вызов `nx.shortest_path_length` в этот раз, результат будет другим, поскольку теперь нельзя перейти напрямую от A к F. Вот новый результат вызова:

```
{'A': 0, 'B': 1, 'C': 2, 'D': 3, 'E': 3, 'F': 4, 'G': 2,
 'H': 1}
```

Обратите внимание, что потеря пути изменила некоторые степени разделения. Теперь расстояние до узла F — самое длинное и равно 4.

Случайное блуждание по графу

Иногда возникает необходимость случайного блуждания по графу. Случайное блуждание по графу, в отличие от поиска конкретного пути, может моделировать естественное поведение, например поиск пищи животным. Этот подход также применим ко многим другим интересным задачам, таким как игры. Однако у случайного блуждания по графу может быть и практическое значение. Например, когда автомобиль застрял в пробке из-за аварии и кратчайший путь больше недоступен. В некоторых случаях выбор случайной альтернативы может оказаться наилучшим решением, поскольку движение по второму кратчайшему маршруту может быть затруднено из-за затора на кратчайшем пути.



Пакет `networkx` не предоставляет прямых средств для получения случайного пути. Однако он позволяет найти все доступные пути, после чего можно случайным образом выбрать один из них. Следующий код демонстрирует, как может работать этот процесс с использованием графа из предыдущего блока:

```
import random
random.seed(0)

paths = nx.all_simple_paths(graph, 'A', 'H')

path_list = []
for path in paths:
    path_list.append(path)
    print("Path Candidate: ", path)
sel_path = random.randint(0, len(path_list) - 1)

print("The selected path is: ", path_list[sel_path])

Path Candidate: ['A', 'B', 'C', 'D', 'E', 'G', 'H']
Path Candidate: ['A', 'H']
Path Candidate: ['A', 'F', 'E', 'G', 'H']
The selected path is: ['A', 'H']
```

В коде устанавливается конкретное начальное значение генератора случайных чисел, чтобы каждый раз получать одинаковый результат. Однако, изменяя это значение, вы можете получать разные результаты из примера кода. Суть в том, что даже простой граф, показанный на рисунке 10.3, предлагает три способа добраться от узла *A* до узла *H* (два из которых определенно длиннее выбранного в данном случае пути). Выбор любого из них гарантирует, что вы доберетесь из одного узла в другой, хотя и потенциально окольным путем.

В ЭТОЙ ГЛАВЕ

- » Почему так сложно найти то, что нужно, в интернете
- » Обзор проблем, которые решает алгоритм PageRank
- » Реализация алгоритма PageRank с учетом телепортации
- » Как меняется использование PageRank

ГЛАВА 11

Поиск нужной веб-страницы

В предыдущих главах мы подробно рассмотрели графы. Веб — один из самых интересных примеров графовой структуры из-за масштаба и сложности взаимосвязей страниц. В этой главе объясняется, почему возможность поиска информации была одновременно важной и сложной задачей и как ранние поисковые системы почти потерпели неудачу, пока не появился Google со своим поисковым алгоритмом PageRank.

После того как мы разобрались с базовыми алгоритмами, позволяющими выполнять обход графов и извлекать полезные структуры (такие, как наличие кластеров или сообществ), эта глава завершает обсуждение графов представлением алгоритма PageRank, который изменил жизнь людей не меньше, чем веб и интернет, сделав веб пригодным для использования. Алгоритм PageRank преобразует ссылки на страницах в рекомендации, подобные рекомендациям экспертов-ученых. Используя примеры и код, глава показывает, как растущий масштаб веба также играет роль в успехе алгоритма. Чем больше веб, тем выше вероятность получить хорошие сигналы для такого умного алгоритма, как PageRank. PageRank — это не только движок Google и многих других поисковых систем, но и умный способ извлечения скрытой информации, такой как релевантность, важность и репутация, из структуры графа (например, при выявлении мошенничества или в природоохранной деятельности).



ЗАПОМНИТЕ

Вам не нужно вводить исходный код для этой главы вручную. На самом деле, использовать загружаемый исходный код намного проще. Вы можете найти исходный код для этой главы в файле **A4D2E; 11; PageRank.ipynb** из загружаемых исходников. Подробнее о том, как найти этот файл, смотрите во введении.

§ 1. Поиск мира в поисковой системе

Библиотеки полагаются на каталоги и библиотекарей, чтобы предложить простой способ найти определенные тексты или изучить определенные темы. Книги не все одинаковы: одни хороши для представления определенных видов информации, другие — лучше. Рекомендации ученых делают книгу авторитетным источником, поскольку эти рекомендации часто появляются в других книгах в виде цитат и ссылок. Такого рода перекрестных ссылок изначально не существовало в вебе. О релевантности веб-страницы судили по наличию определенных слов в заголовке или в тексте. Этот подход был практически равносителен оценке книги по ее названию и количеству содержащихся в ней слов.

Для многих людей сегодня их личная и профессиональная жизнь немислима без интернета и веба. Сеть Интернет состоит из взаимосвязанных страниц (помимо прочего). Веб состоит из сайтов, доступных по доменам, каждый из которых состоит из страниц и гиперссылок, соединяющих сайты внутренне и с другими сайтами внешне. Сервисы и информационные ресурсы доступны через веб, если вы точно знаете, где искать. Доступ к вебу немислим без поисковых систем — сайтов, которые позволяют найти что угодно в вебе с помощью простого запроса.

Поиск данных в интернете

По оценкам, размер веба колеблется между 35—40 миллиардами страниц (<https://www.worldwidewebsize.com>), поэтому его непросто представить в виде графа. Исследования описывают веб как граф в форме галстука-бабочки (см. на <https://immorlica.com/socNet/broder.pdf>). Веб в основном состоит из взаимосвязанного ядра и других частей, которые ссылаются на это ядро. С его первого расширения в начале 1990-х годов многие отмечали проблему в оценке формы и размера веба. Кроме того, обходить его для поиска информации было непросто: многие его части были разрознены или труднодоступны.

Поисковые системы были разработаны, чтобы упростить работу с необычным размером и формой веба — сделать его доступным и полезным для всех. Двигаясь по любой дороге в реальном мире, вы можете добраться куда угодно (возможно, придется пересечь океаны). Однако в вебе нельзя достичь всех сайтов, просто следуя его структуре, — некоторые части недоступны (они отключены или вы не можете до них добраться). Если хотите что-то найти в вебе, даже когда время не проблема, вам все равно нужен индекс.

Как найти нужные данные

Поиск нужных данных был проблемой с ранних лет существования веба, но первые поисковые системы появились только в 1990-х годах. О поисковых системах не думали раньше, поскольку другие решения, такие как простые списки доменов или специализированные каталоги сайтов, работали хорошо. Только когда эти решения перестали масштабироваться из-за быстрорастущего размера веба, появились такие поисковые системы, как Lycos, Magellan, Yahoo!, Excite, Inktomi и AltaVista.

Все эти поисковые системы работали благодаря специализированному программному обеспечению, которое автономно посещало веб, используя списки доменов и проверяя гиперссылки, найденные на посещенных страницах. Эти «пауки» исследовали каждую новую ссылку в процессе, называемом *обходом*. «Пауки» — это программы, которые читают страницы как обычный текст (они не могут понимать изображения или другой нетекстовый контент).

Ранние поисковые системы работали путем обхода веба, сбора информации от «пауков» и ее обработки для создания инвертированных индексов. Индексы позволяли находить страницы на основе содержащихся в них слов. Когда вы делали запрос, такие инвертированные индексы сообщали обо всех страницах, содержащих искомые термины, и помогали оценивать страницы, создавая рейтинг, который превращался в результат поиска. Результатом был упорядоченный список страниц, начиная с предположительно самой полезной страницы и заканчивая наименее полезной.

Система оценки была довольно примитивной, поскольку учитывала, как часто ключевые слова встречались на страницах или присутствовали ли они в заголовках страниц. Иногда ключевые слова получали более высокий балл, если были перемешаны или сгруппированы. Такие простые методы индексации и оценки позволяли некоторым пользователям интернета прибегать к различным уловкам. Среди них:

- » **веб-спамеры** – использовали свои возможности, чтобы заполнять результаты поиска страницами с некачественным контентом и большим количеством рекламы;
- » **неэтичная поисковая оптимизация** – применялась людьми, которые использовали свои знания о поисковых системах, чтобы искусственно повысить позиции манипулируемых ими страниц в результатах поиска, несмотря на их низкое качество. К сожалению, эти проблемы сохраняются и сегодня, поскольку поисковые системы, даже самые продвинутые, не защищены от людей, желающих обмануть систему для получения более высоких позиций в поисковой выдаче. Алгоритм PageRank может устранить многих старых спамеров и неэтичных SEO-специалистов, но это не панацея.



ЗАПОМНИТЕ

Важно различать неэтичную и этичную SEO-оптимизацию. Специалисты, практикующие этичную SEO-оптимизацию, — это профессионалы, которые используют свои знания о поисковых системах для лучшего продвижения действительно полезных страниц законными и этичными способами.

Появление таких игроков и возможность манипулировать результатами поисковых систем создали потребность в более совершенных алгоритмах ранжирования. Одним из таких решений стал алгоритм PageRank.

§ 2. Объяснение алгоритма PageRank

Алгоритм PageRank назван в честь сооснователя Google Ларри Пейджа. Впервые алгоритм был представлен публично в 1998 году в статье «Анатомия крупномасштабной гипертекстовой поисковой системы», написанной Сергеем Брином и Ларри Пейджем и опубликованной в журнале *Computer Networks and ISDN Systems* (<http://ilpubs.stanford.edu:8090/361/1/1998-8.pdf>). В то время оба, Брин и Пейдж, были аспирантами, а алгоритм, ставший фундаментом поисковой технологии Google, изначально был исследовательским проектом в Стэнфордском университете.

Говоря простым языком, *PageRank* оценивает важность каждого узла в графе таким образом, что чем выше оценка, тем важнее узел в графе. Определение важности узла в таком графе, как веб, означает вычисление релевантности страницы как части результатов поискового запроса, что позволяет лучше обслуживать пользователей, ищущих качественный веб-контент.

Страница считается хорошим ответом на запрос, когда она соответствует критериям запроса и значима в системе гиперссылок, связывающих страницы между собой. Логика, лежащая в основе значимости, заключается в том, что, поскольку веб создается пользователями, страница имеет важность в сети по веской причине (качество и авторитетность контента страницы оценивается по ее важности в сетевой структуре веб-гиперссылок).

Понимание логики алгоритма PageRank

В 1998 году, когда Брин и Пейдж еще были студентами Стэнфордского университета, качество результатов поиска было проблемой для всех пользователей интернета. Популярные поисковые системы того времени испытывали трудности как с постоянно растущей структурой веба (в разделе 4 обсуждаются проблемы масштабирования алгоритмов и способы их работы с большими данными), так и со множеством спамеров.



ЗАПОМНИТЕ

Термин *спамеры* в данном случае относится не к тем, кто рассылает нежелательные письма на электронную почту, а к веб-спамерам — тем, кто осознавал экономическую важность размещения страниц на верхних позициях результатов поиска. Эта группа разработала сложные и злонамеренные приемы для обмана поисковых систем. Популярные хитрости веб-спамеров того времени включали:

- » **переполнение ключевыми словами** – подразумевает чрезмерное использование определенных ключевых слов на странице, чтобы обмануть поисковую систему, заставив ее думать, что страница серьезно обсуждает тему этих ключевых слов;
- » **невидимый текст** – предполагает копирование содержимого страницы-результата по запросу на другую страницу с использованием одинакового цвета для символов и фона. Скопированный контент невидим для пользователей, но виден поисковым роботам (которые тогда, как и сейчас, просто сканировали текстовые данные) и их алгоритмам. Этот трюк позволял странице с невидимым текстом получить такой же высокий рейтинг в поиске, как и у исходной страницы;
- » **клоакинг** – представляет собой более сложный вариант невидимого текста, когда вместо текста скрипты или изображения предоставляют поисковым роботам контент, отличающийся от того, что видят пользователи.

Веб-спамеры использовали такие уловки, чтобы обманом получить высокие позиции в поисковых системах, даже если содержание страницы было некачественным и в лучшем случае вводящим в заблуждение. Эти трюки имели последствия. Например, пользователь мог искать информацию, связанную с университетскими исследованиями, а вместо этого сталкивался с коммерческой рекламой или неподходящим контентом. Пользователи были разочарованы, поскольку часто попадали на страницы, не связанные с их потребностями, что вынуждало их переформулировать запросы и тратить время на поиск полезной информации среди страниц результатов, впустую расходуя силы на отделение хороших источников от плохих. Ученые и эксперты, отмечая необходимость борьбы со спам-результатами и опасаясь, что развитие веба может остановиться из-за трудностей пользователей в поиске действительно нужной информации, начали работать над возможными решениями.

Пока Брин и Пейдж работали над своим алгоритмом, параллельно разрабатывались и публиковались другие идеи. Одной из таких идей был Hyper Search Массимо Маркиори, который первым указал на важность веб-ссылок при определении значимости веб-страницы как фактора, который следует учитывать при поиске: <https://www.w3.org/People/Massimo/papers/WWW6>. Другим интересным решением стал проект поисковой системы под названием HITS (Hypertext-Induced Topic Search, «Поиск по гипертекстовым темам»), также основанный на структуре веб-ссылок и разработанный Джоном Клейнбергом, молодым ученым, работавшим в IBM Almaden в Силиконовой долине. Интересный факт о HITS заключается в том, что он классифицирует страницы на *хабы* (страницы со множеством ссылок на авторитетные страницы) и *авторитетные страницы* (страницы, считающиеся авторитетными благодаря множеству ссылок с хабов) — то, что PageRank не делает явно (хотя неявно учитывает в вычислениях) (<http://pi.math.cornell.edu/~mec/Winter2009/RalucaRemus/Lecture4/lecture4.html>).



ЗАПОМНИТЕ

Когда приходит время, одна и та же идея или что-то похожее часто прорастает в разных местах. Иногда происходит обмен базовыми идеями между исследователями и учеными; иногда идеи развиваются независимо (см. историю японского математика Такакадзу Сэки на <https://mathshistory.st-andrews.ac.uk/Biographies/Seki>, который независимо открыл многое из того, что и европейские математики, такие как Ньютон, Лейбниц и Бернулли, примерно в тот же период). В 1998 году только Брин и Пейдж предприняли шаги к созданию поисковой компании на основе своего алгоритма, взяв академический отпуск в Стэнфордском университете и прервав работу над докторскими диссертациями, чтобы сосредоточиться на одном: заставить свой алгоритм работать более чем с миллиардом веб-страниц.

Объясняя принципы работы PageRank

Инновация, принесенная PageRank, заключается в том, что инвертированного индекса терминов недостаточно для определения, соответствует ли страница информационному запросу пользователя. Совпадение слов (или смысла; семантическое соответствие запросу, о котором говорится в конце главы) между запросом и текстом страницы — необходимое условие, но недостаточное, поскольку гиперссылки нужны для оценки, предлагает ли страница качественный контент и является ли авторитетной.

При обсуждении сайтов важно различать входящие и исходящие ссылки, при этом не следует учитывать внутренние ссылки, соединяющие страницы в пределах одного сайта. Ссылки, которые вы видите на странице, являются *исходящими*, когда они ведут на другую страницу другого сайта. Ссылки, которые приводят посетителей на вашу страницу с других сайтов, называются *входящими* (обратными). Как создатель страницы, вы используете исходящие ссылки, чтобы предоставить дополнительную информацию о содержимом страницы. Вряд ли вы будете размещать случайные ссылки на своей странице (или ссылки, ведущие на бесполезный или некачественный контент), поскольку это снизит качество страницы. Когда вы ссылаетесь на качественный контент, другие создатели страниц тоже ставят ссылки на вашу страницу, если она интересна и высококачественная.

Это цепочка доверия. Гиперссылки подобны рекомендациям или одобрению страниц. Входящие ссылки показывают, что другие создатели страниц доверяют вам, а вы делитесь частью этого доверия, добавляя исходящие ссылки на своих страницах, указывающие на другие ресурсы.

§ 3. Реализация PageRank

Для математического представления этой цепочки доверия нужно одновременно определить, какой авторитет у вашей страницы (измеряемый входящими ссылками) и сколько она передает другим страницам через исходящие ссылки. Такие вычисления можно выполнить двумя способами:

- » **симуляцией** — использует поведение веб-серфера, который случайным образом перемещается по сети (*случайный серфер*). Этот подход требует воссоздания структуры веба и запуска симуляции;

» **матричными вычислениями** – воспроизводят поведение случайного серфера с помощью *разреженной матрицы* (матрицы, в которой большинство данных равно нулю), отражающей структуру веба. Этот подход требует некоторых матричных операций, как объясняется в главе 5, и серии вычислений, которые достигают результата путем последовательных приближений.

Несмотря на большую абстрактность, использование матричных вычислений для PageRank требует меньше программных инструкций и его легко реализовать на Python. (Вы можете опробовать алгоритм PageRank на реальных сайтах с помощью автоматического проверщика PageRank, например: <https://checkpagerank.net/index.php>. К сожалению, программа может выдавать неточные результаты для новых сайтов, поскольку они еще не были должным образом проиндексированы, но это даст вам представление о том, как PageRank работает на практике.)

Реализация скрипта на Python

PageRank — это функция, которая оценивает узлы в графе числом (чем выше число, тем важнее узел). При оценке веб-страницы это число может представлять вероятность посещения случайным серфером. Вероятности выражаются числом от 0,0 до максимум 1,0; в идеале, когда представляется вероятность нахождения на определенном сайте среди всех доступных сайтов, сумма всех вероятностей страниц в сети должна равняться 1,0.



ЗАПОМНИТЕ

Есть много версий PageRank. Каждая из них немного меняет свой рецепт в соответствии с типом графа, который нужно оценить. Пример в этом блоке представляет вам оригинальную версию для веба, описанную в ранее упомянутой статье Брина и Пейджа, а также в статье «The PageRank Citation Ranking: Bringing Order to the Web» (<http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf>).

В примере создаются три различные веб-сети, состоящие из шести узлов (веб-страниц). Первая представляет собой корректно работающую сеть, а две другие демонстрируют проблемы, с которыми может столкнуться случайный пользователь из-за структуры веб-сети или действий веб-спамеров. В этом примере также используются команды NetworkX, рассмотренные в главе 8, и пакет NumPy для матричных вычислений, описанный в главе 4.

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
```

```

Graph_A = nx.DiGraph()
Graph_B = nx.DiGraph()
Graph_C = nx.DiGraph()
Nodes = range(1,6)
Edges_OK = [(1,2),(1,3),(2,3),(3,1),(3,2),(3,4),(4,5),
            (4,6),(5,4),(5,6),(6,5),(6,1)]
Edges_dead_end = [(1,2),(1,3),(3,1),(3,2),(3,4),(4,5),
                  (4,6),(5,4),(5,6),(6,5),(6,1)]
Edges_trap = [(1,2),(1,3),(2,3),(3,1),(3,2),(3,4),(4,5),
              (4,6),(5,4),(5,6),(6,5)]
Graph_A.add_nodes_from(Nodes)
Graph_A.add_edges_from(Edges_OK)
Graph_B.add_nodes_from(Nodes)
Graph_B.add_edges_from(Edges_dead_end)
Graph_C.add_nodes_from(Nodes)
Graph_C.add_edges_from(Edges_trap)

```

Этот код отображает первую сеть — работающую правильно, как показано на рисунке 11.1.

```

np.random.seed(2)
pos=nx.shell_layout(Graph_A)
draw_params = {'with_labels':True,
               'arrows': True,
               'node_color':'skyblue',
               'node_size':700, 'width':2,
               'font_size':14}
nx.draw(Graph_A, pos, **draw_params)
plt.show()

```

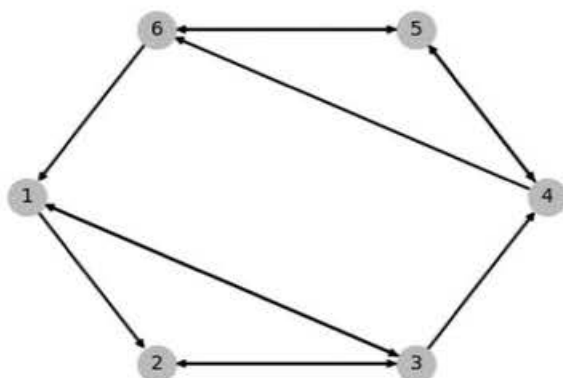


РИСУНОК 11.1.
Сильносвязанная сеть



СОВЕТ

Все узлы соединены друг с другом. Это пример сильносвязанного графа, который не содержит изолированных узлов, одиночных вершин и анклавов, действующих как тупики. Случайный пользователь может свободно перемещаться по нему, никогда не останавливаясь, и любой узел может достичь любого другого узла. В представлении направленного графа в NetworkX используются стрелки для обозначения направления ребра. Например, пользователь может перейти из узла 4 в узел 6, поскольку существует жирная линия, входящая в узел 6 из узла 4. Однако пользователь не может перейти из узла 6 в узел 4, поскольку линия, входящая в узел 4 из узла 6, тонкая.

Второй граф не является сильносвязанным. Он представляет собой ловушку для случайного пользователя, так как у узла 2 нет исходящих связей и пользователь, посетивший страницу, может остановиться там и не найти выхода. Это не является необычным событием, учитывая структуру веб-сети, но может указывать на действия спамера, который создал «фабрику спама» со множеством ссылок, ведущих на тупиковый сайт для захвата веб-пользователей. На рисунке 11.2 показан результат выполнения следующего кода, который использовался для отображения этого графа:

```
np.random.seed(2)
pos=nx.shell_layout(Graph_B)
nx.draw(Graph_B, pos, **draw_params)
plt.show()
```

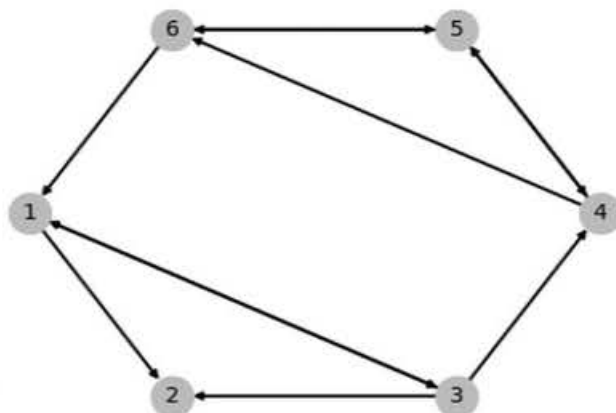


РИСУНОК 11.2.
Сеть с тупиком
в узле 2

Другая ситуация, которая может быть естественной или результатом действий спамера, — это паучья ловушка. Это еще один тупик для пользователя, на этот раз не на одной странице, а на закрытом сайте, где нет ссылок на внешнюю сеть страниц. На рисунке 11.3 показан результат выполнения следующего кода, который использовался для отображения этого графа:


```
np.random.seed(2)
pos=nx.shell_layout(Graph_C)
nx.draw(Graph_C, pos, **draw_params)
plt.show()
```

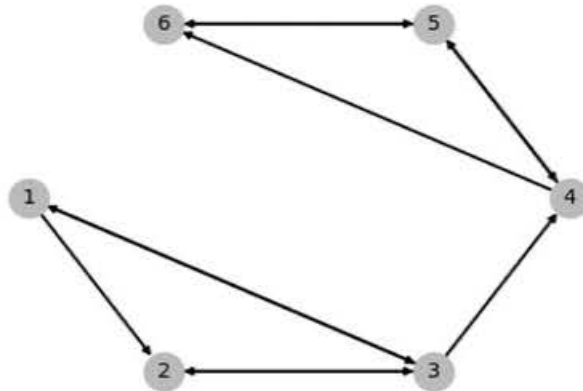


РИСУНОК 11.3.
Сеть с паучьей
ловушкой в уз-
лах 4, 5 и 6



ЗАПОМНИТЕ

Это называется *паучьей ловушкой*, потому что спамеры придумали ее как способ поймать поисковых роботов в замкнутый цикл, заставляя их думать, что существуют только те сайты, которые находятся внутри закрытой сети.

Борьба с наивной реализацией

Имея граф, созданный с помощью Python и NetworkX, можно извлечь его структуру и представить в виде *матрицы переходов* — матрицы, которая отображает узлы в столбцах и строках.

- » **Столбцы** – содержат узел, на котором находится веб-сервер.
- » **Строки** – содержат вероятность того, что сервер посетит другие узлы по исходящим ссылкам.

В реальном интернете матрица переходов, которая используется алгоритмом PageRank, создается путем непрерывного исследования ссылок поисковыми роботами.

```
def initialize_PageRank(graph):
    nodes = len(graph)
    M = nx.to_numpy_matrix(graph)
    outbound = np.squeeze(np.asarray(np.sum(M, axis=1)))
    prob_outbound = np.array(
```

```

        [1.0/count
         if count>0 else 0.0 for count in outbound])
    G = np.asarray(np.multiply(M.T, prob_outbound))
    p = np.ones(nodes) / float(nodes)
    if np.min(np.sum(G,axis=0)) < 1.0:
        print ('Warning: G is substochastic')
    return G, p

```

Python-код создает функцию `initialize_PageRank()`, которая извлекает как матрицу переходов, так и начальный вектор стандартных оценок PageRank.

```

G, p = initialize_PageRank(Graph_A)
print(G)

[[ 0.    0.  0.33333333  0.    0.    0.5 ]
 [ 0.5  0.  0.33333333  0.    0.    0. ]
 [ 0.5  1.  0.          0.    0.    0. ]
 [ 0.    0.  0.33333333  0.    0.5  0. ]
 [ 0.    0.  0.          0.5  0.    0.5 ]
 [ 0.    0.  0.          0.5  0.5  0. ]]

```

Напечатанная матрица переходов **G** представляет матрицу переходов сети, описанной на рисунке 11.1. Каждый столбец представляет узел в последовательности от 1 до 6. Например, третий столбец представляет узел 3. Каждая строка в столбце показывает связи с другими узлами (исходящие ссылки на узлы 1, 2 и 4) и значения, определяющие вероятность того, что случайный серфер воспользуется любой из исходящих ссылок (то есть $1/3$, $1/3$, $1/3$).



СОВЕТ

Диагональ матрицы всегда равна нулю, если только у страницы нет исходящей ссылки на саму себя (что возможно).

Матрица содержит больше нулей, чем значащих элементов. Это соответствует реальности, поскольку, по оценкам, у каждого сайта есть в среднем только десять исходящих ссылок. Поскольку существуют миллиарды сайтов, ненулевые значения в матрице переходов, представляющей веб, минимальны. В этом случае полезно использовать такую структуру данных, как список смежности (описанную в блоке «Создание графа» главы 9 и упомянутую снова в блоке «Распределение файлов и операций» главы 13), чтобы представлять данные без траты дискового пространства или памяти на нулевые значения:

```

from scipy import sparse
sG = sparse.csr_matrix(G)
print (sG)

```

```
(0, 2) 0.333333333333
(0, 5) 0.5
(1, 0) 0.5
(1, 2) 0.333333333333
(2, 0) 0.5
(2, 1) 1.0
(3, 2) 0.333333333333
(3, 4) 0.5
(4, 3) 0.5
(4, 5) 0.5
(5, 3) 0.5
(5, 4) 0.5
```

В этом примере всего 12 ссылок из 30 возможных (не считая ссылок *на самого себя*, то есть на текущий сайт). Другая особенность матрицы переходов, которую следует отметить, — то, что сумма значений в столбцах должна равняться 1,0. Если она меньше 1,0, матрица является *субстохастической* (что означает, что данные матрицы неправильно представляют вероятности, поскольку вероятности должны в сумме давать 1,0) и не может корректно работать с оценками PageRank.

Вместе с **G** существует вектор **p** — начальная оценка общего показателя PageRank, равномерно распределенная между узлами. В этом примере, поскольку общий PageRank равен 1,0 (вероятность нахождения случайного пользователя в сети составляет 100%), он распределяется как 1/6 между шестью узлами.

```
print(p)

[ 0.16666667  0.16666667  0.16666667  0.16666667
  0.16666667  0.16666667]
```

Чтобы оценить PageRank, возьмите начальную оценку для узла в векторе **p**, умножьте ее на соответствующий столбец в матрице переходов и определите, какая часть его PageRank (его авторитетности) передается другим узлам. Повторите это для всех узлов, и вы узнаете, как PageRank передается между узлами из-за структуры сети. Этого вычисления можно достичь с помощью умножения матрицы на вектор.

```
print(np.dot(G, p))

[ 0.13888889  0.13888889  0.25      0.13888889
  0.16666667  0.16666667]
```


После первого умножения матрицы на вектор вы получаете другую оценку PageRank, которую используете для перераспределения между узлами. При многократном перераспределении оценка PageRank стабилизируется (результаты перестают меняться), и вы получите нужные значения. Использование матрицы переходов, содержащей вероятности, и последовательное приближение с помощью умножения матрицы на вектор дают те же результаты, что и компьютерное моделирование со случайным пользователем:

```
def PageRank_naive(graph, iters = 50):
    G, p = initialize_PageRank(graph)
    for i in range(iters):
        p = np.dot(G,p)
    return np.round(p,3)

print(PageRank_naive(Graph_A))

[ 0.154  0.154  0.231  0.154  0.154  0.154]
```

Новая функция `PageRank_naive()` объединяет все ранее описанные операции и выдает вектор вероятностей (оценку PageRank) для каждого узла в сети. Узел 3 оказывается наиболее важным. К сожалению, эта же функция не работает с двумя другими сетями:

```
print(PageRank_naive(Graph_B))
Warning: G is substochastic
[ 0.  0.  0.  0.  0.  0.]

print(PageRank_naive(Graph_C))
[ 0.  0.  0.  0.222  0.444  0.333]
```

В первом случае вероятности, похоже, «утекают» из сети — это эффект тупикового веб-сайта и получающейся в результате субстохастической матрицы переходов. Во втором случае нижняя половина сети несправедливо получает всю значимость, оставляя верхнюю часть незначительной.

Внедрение скуки и телепортации

Как тупики (*стоки ранга*), так и паузы ловушки (*циклы*) — распространенные ситуации в интернете из-за действий пользователей и спамеров. Однако проблема легко решается, если заставить случайного пользователя случайным образом перепрыгивать на другой узел сети (*телепортироваться*, как в научно-фантастических устройствах, мгновенно переносящих вас из одного места в другое). Теория заключается в том, что пользователь заскучает и уйдет из тупиковых ситуаций. Математически вы определяете значение α , представляющее вероятность того, что пользователь продолжит случайное путешествие по графу. Значение α перераспределяет вероятность нахождения в узле независимо от матрицы переходов.



СОВЕТ

Значение α (также называемое фактором затухания), изначально предложенное Брином и Пейджем, равно 0,85 (или только 0,15 вероятности телепортации), но вы можете изменить его в соответствии с вашими потребностями. Для веба оно лучше всего работает между 0,8 и 0,9 (о причинах можно прочитать на <https://www.cise.ufl.edu/~adobra/DaMn/talks/damn05-santini.pdf>). Чем меньше значение α , тем короче в среднем путешествие пользователя по сети перед перезапуском где-то еще.

```
def PageRank_teleporting(graph, iters=50, alpha=0.85,
                        rounding=3):
```

```
    G, p = initialize_PageRank(graph)
```

```
    u = np.ones(len(p)) / float(len(p))
```

```
    for i in range(iters):
```

```
        p = alpha * np.dot(G,p) + (1.0 - alpha) * u
```

```
    return np.round(p / np.sum(p), rounding)
```

```
print('Graph A:', PageRank_teleporting(Graph_A,
                                       rounding=8))
```

```
print('Graph B:', PageRank_teleporting(Graph_B,
                                       rounding=8))
```

```
print('Graph C:', PageRank_teleporting(Graph_C,
                                       rounding=8))
```

```
Graph A: [ 0.15477863  0.15346061  0.22122243  0.15477863
           0.15787985  0.15787985]
```

```
Warning: G is substochastic
```

```
Graph B: [ 0.16502904  0.14922238  0.11627717  0.16502904
           0.20222118  0.20222118]
```

```
Graph C: [ 0.0598128  0.08523323  0.12286869  0.18996342
           0.30623677  0.23588508]
```

После применения модификаций к новой функции `PageRank_teleporting()` вы можете получить похожие оценки для первого графа, а еще гораздо более реалистичные (и полезные) оценки для второго и третьего графа, не попадая в ловушки тупиков или стоков ранга. Интересно, что функция эквивалентна той, что предоставляется NetworkX: https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.link_analysis.pagerank_alg.pagerank.html.

```
nx.pagerank(Graph_A, alpha=0.85)
```

```
{1: 0.15477892494151968,  
2: 0.1534602056628941,  
3: 0.2212224378270561,  
4: 0.15477892494151968,  
5: 0.1578797533135051,  
6: 0.15787975331350507}
```

Заглядывая внутрь поисковой системы

Хотя PageRank анализирует только структуру гиперссылок в интернете, он показывает, насколько авторитетной может стать веб-страница. Однако алгоритм ранжирования Google состоит не только из PageRank. Этот алгоритм создает прочную основу для любого поискового запроса, и изначально именно он принес Google славу надежной поисковой системы. Сегодня PageRank лишь один из множества факторов ранжирования, участвующих в обработке запроса.

Специализированные источники в области SEO называют более 200 факторов, влияющих на результаты поиска Google. Чтобы узнать, какие еще факторы ранжирования учитывает Google, ознакомьтесь со списками на <https://moz.com/learn/seo/on-page-factors> и <https://moz.com/learn/seo/off-site-seo> (составлены американской компанией MOZ).

Важно также учитывать, что алгоритм Google получил множество обновлений, и сейчас это скорее ансамбль различных алгоритмов, каждому из которых присвоено фантазийное название (Caffeine, Panda, Penguin, Hummingbird, Pigeon, Mobile Update). Полный, упорядоченный по годам список можно найти на <https://www.searchenginejournal.com/google-algorithm-history> благодаря SEJ — SearchEngine Journal. Многие из этих обновлений вызвали серьезные изменения в предыдущих поисковых рейтингах и были мотивированы необходимостью бороться со спам-техниками или сделать веб-серфинг более удобным для пользователей; например, Mobile Update побудил многие сайты адаптировать свои интерфейсы под мобильные телефоны.

Другие применения PageRank

Хотя PageRank обеспечивает лучшие результаты поиска, его применимость не ограничивается Google или поисковыми системами. Вы можете использовать PageRank везде, где можете свести свою задачу к графу. Просто модифицируйте и настройте алгоритм под свои нужды. Корнельский университет перечислил некоторые другие потенциальные применения PageRank в различных областях (<https://blogs.cornell.edu/info2040/2014/11/03/more-than-just-a-web-search-algorithm-googles-pagerank-in-non-internet-contexts>), а также появились удивительные сообщения об успешном использовании алгоритма в вычислительной биологии (<https://www.wired.com/2009/09/googlefoodwebs>). Создавая телепортацию, привязанную к конкретным узлам, которые вы хотите исследовать, вы увидите, как алгоритм блестяще проявляет себя в различных приложениях, таких как:

- » **выявление мошенничества** – обнаружение неожиданных связей между определенными людьми и фактами;
- » **рекомендация товаров** – предложение продуктов, которые могут понравиться человеку с определенными предпочтениями.

§ 4. Выход за рамки парадигмы PageRank

В последние годы Google не только ввел дополнительные факторы ранжирования, модифицирующие исходный алгоритм PageRank. Компания внедрила радикальные изменения, которые лучше используют содержимое страниц (чтобы не попадаться на присутствие определенных ключевых слов), и применила алгоритмы искусственного интеллекта, которые автономно оценивают релевантность страницы в результатах поиска. Эти изменения привели к тому, что некоторые эксперты по поиску заявили, что PageRank больше не определяет позицию страницы в поиске (см. на <https://www.entrepreneur.com/article/269574>). Дискуссии по этому вопросу продолжаются, но, вероятнее всего, можно предположить, что PageRank все еще работает в поисковой системе Google как фактор ранжирования, хотя и недостаточный для того, чтобы вывести страницу в лучшие результаты после запроса.

Знакомство с семантическими запросами

Если сейчас вы попытаетесь задать Google вопросы, а не просто цепочки ключевых слов, то заметите, что он стремится отвечать разумно и улавливает смысл вопроса. С 2012 года Google стал лучше понимать синонимы и концепции. Однако после августа 2013 года обновление Hummingbird (<https://searchengineland.com/google-hummingbird-172816>) действительно изменило правила игры, поскольку поисковая система стала способна понимать разговорные поисковые запросы (запросы, в которых вы спрашиваете что-то так, как сказали бы это другому человеку), а еще семантику, стоящую за запросами и содержанием страниц.

После этого обновления алгоритм Google стал работать путем распознавания как намерений пользователей, так и смыслов, выраженных на страницах, а не только по ключевым словам. Обновление сделало поисковую систему более семантической, что означает понимание того, что подразумевают слова с обеих сторон: в запросе и на найденных веб-страницах. В этом смысле его больше нельзя обмануть, манипулируя ключевыми словами. Даже без существенной поддержки PageRank поисковая система может анализировать, как написана страница, и определять, содержит ли она контент, достаточно качественный для включения в результаты поиска.

Использование ИИ для ранжирования результатов поиска

PageRank по-прежнему остается основой ранжирования результатов поиска, но его влияние уменьшилось из-за внедрения технологий машинного обучения в ранжирование, так называемого RankBrain. Согласно некоторым источникам (см. на <https://www.searchenginejournal.com/google-algorithm-history/rankbrain>), алгоритм машинного обучения теперь действует как предобработчик — фильтрующее устройство, которое оценивает полученный запрос и вычисляет соответствующие веса для сигналов, которые он обнаруживает в запросе.

Хотя RankBrain все еще окутан тайной, его значение возросло, и алгоритм, похоже, способен угадывать с гораздо большей точностью, чем человек, может ли содержимое страницы появиться в результатах поиска. Он переопределяет все остальные факторы ранжирования в случаях, которые трудно оценить. Это еще один пример дополнительного алгоритма, ограничивающего роль, которую играет оригинальный алгоритм PageRank.



**ОБРАБОТКА
БОЛЬШИХ
ДАННЫХ**

В ЭТОМ РАЗДЕЛЕ

Взаимодействие с большими данными
и управление ими

Более эффективная обработка больших
данных

Уменьшение объемов данных

Соккрытие данных с помощью шифрования

В ЭТОЙ ГЛАВЕ

- » Знакомство с законом Мура и его последствиями
- » Понимание больших данных и их 4V
- » Изучение способов работы с бесконечными потоковыми данными
- » Использование выборки, хеширования и эскизов (sketches) для обработки потоковых данных

ГЛАВА 12

Управление большими данными

Большие данные — это больше, чем модное словечко, используемое вендорами для продвижения инновационных способов хранения и анализа данных. Большие данные также не являются преходящим увлечением. Напротив, это реальность и движущая сила нашего времени. Вы могли слышать об этом во многих специализированных научных и деловых публикациях и даже задаваться вопросом, что на самом деле означает «большие данные» (нет, это не про размещение информации на рекламных щитах большими буквами). С технической точки зрения большие данные относятся к большим и сложным массивам компьютерных данных — настолько большим (как следует из названия) и запутанным, что с ними нельзя справиться, просто увеличив объем хранилища на компьютерах или сделав новые компьютеры более мощными и быстрыми в вычислениях. В первом параграфе этой главы обсуждаются последствия больших данных и их значение в отношении алгоритмов.

Вам нужно знать о данных больше, чем просто их происхождение или влияние на аппаратное и программное обеспечение и на анализ. Второй параграф главы рассматривает концепцию *потоковых данных*, которая включает приведение их в форму для упорядоченного просмотра. Это означает применение таких алгоритмических инструментов, как выборка и хеширование, о которых вы можете прочитать во втором параграфе этой

главы. Вы также рассмотрите эффекты *выборки* — процесса резервирования нужных данных для различных целей. (Хеширование подробно рассматривается в параграфе «Опираясь на хеширование» главы 7.)

В третьем параграфе главы рассматривается *скетчинг* — метод создания простого и приближенного обзора данных. Этот процесс начинается с хеширования, а затем переходит к созданию сводки данных с использованием различных алгоритмов.



ЗАПОМНИТЕ

Вам не нужно вручную набирать исходный код для этой главы. На самом деле, использовать загружаемый исходный код намного проще. Исходный код главы можно найти в файле `\A4D2E\A4D2E; 12; Managing Big Data.ipynb` из загружаемых материалов. Подробнее о том, как найти эти исходные файлы, смотрите во введении.

§ 1. Преобразование энергии в данные

В 1965 году Гордон Мур, соучредитель компаний Intel и Fairchild Semiconductor (двух гигантских компаний, производящих электронные компоненты для электроники и компьютеров), в статье журнала *Electronics* под названием «Размещение большого количества компонентов на интегральных схемах» заявил, что количество компонентов в интегральных схемах будет удваиваться каждый год в течение следующего десятилетия (<https://www.britannica.com/technology/Moores-law>). (Позже он пересмотрел свой прогноз до каждых двух лет.) В то время в электронике доминировали транзисторы. Возможность размещения большого количества транзисторов в схеме с помощью единого электронного компонента, объединяющего функциональность многих из них (интегральной схемы), означала возможность сделать электронные устройства более мощными и полезными. Этот процесс называется *интеграцией* и подразумевает активную миниатюризацию электроники (создание той же схемы в гораздо меньшем размере, что логично, поскольку тот же объем должен вмещать вдвое больше схем в сравнении с объемом предыдущего года).

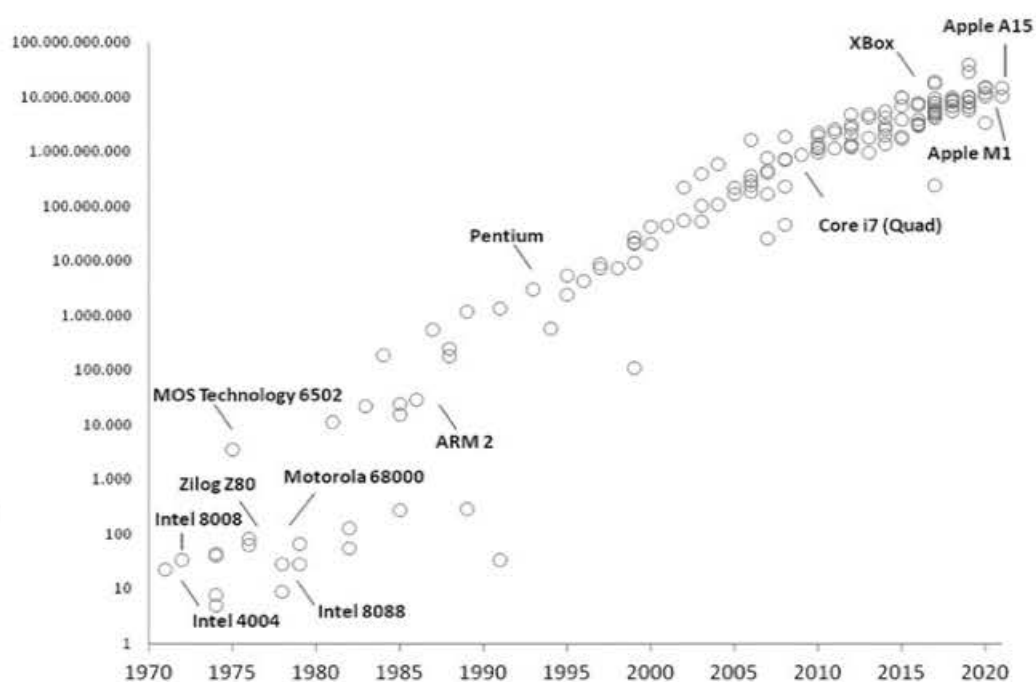
По мере развития миниатюризации электронные устройства — конечный продукт этого процесса — становятся либо меньше, либо просто мощнее. Например, современные компьютеры не больше компьютеров десятилетней давности, но определенно мощнее. То же самое касается и мобильных телефонов. Хотя у них те же размеры, что и у их предшественников, они стали способны выполнять больше задач. Другие устройства, такие как датчики, просто стали меньше, что позволяет размещать их повсюду и находить новые, непредвиденные ранее способы применения.

Понимание следствий закона Мура

То, что Мур изложил в статье, упомянутой в начале параграфа, оставалось верным на протяжении многих лет, и полупроводниковая индустрия называет это законом Мура (подробнее см. на <http://www.moorelaw.org>). В первые десять лет удвоение действительно происходило, как и было предсказано. В 1975 году Мур скорректировал свое утверждение, спрогнозировав удвоение каждые два года. На рисунке 12.1 показаны эффекты этого удвоения. Такой темп удвоения все еще сохраняется (<https://www.wired.com/beyond-the-beyond/2020/03/preparing-end-moores-law>), хотя некоторые скептики недовольны этим и предрекают его конец (другие с этим не согласны). Начиная с 2012 года возникает несоответствие между ожиданиями по размещению большего количества транзисторов в компоненте для повышения его производительности и тем, чего полупроводниковые компании могут реально достичь в плане миниатюризации. По правде говоря, существуют физические барьеры для интеграции большего количества схем в интегральную микросхему с использованием современных кремниевых компонентов. (Тем не менее инновации будут продолжаться; подробнее об этом можно прочитать в статье на <https://www.nature.com/news/the-chips-are-down-for-moore-s-law-1.19338>). Кроме того, закон Мура на самом деле не закон — это наблюдение или даже предварительная, неявная цель, к достижению которой стремится индустрия (в некотором смысле самосбывающееся пророчество).

РИСУНОК 12.1.

Размещение все большего количества транзисторов в процессоре



В итоге закон Мура может перестать действовать, поскольку индустрия перейдет на новые технологии, такие как создание компонентов с использованием оптических лазеров вместо транзисторов (подробности об оптических вычислениях см. в статье на <https://sciencenordic.com/computers-denmark-forskerzonen/optical-computers-light-up-the-horizon/1454763>), или обратится к квантовым вычислениям (см. статью на <https://www.nasa.gov/feature/ames/quantum-supremacy>). Важно то, что с 1965-го примерно каждые два года компьютерная индустрия переживала значительные достижения в области цифровой электроники, которые имели серьезные последствия.



ЗАПОМНИТЕ

Некоторые утверждают, что закон Мура уже перестал действовать. Хотя полупроводниковая индустрия до сих пор выполняла обещанное, сейчас она снижает ожидания. Intel уже увеличила интервалы между поколениями своих процессоров, заявляя, что через пять лет миниатюризация чипов достигнет предела. (Подробнее о волнах в развитии чипов можно прочитать по ссылке <https://scitechdaily.com/twilight-for-silicon-end-of-moores-law-in-view-as-silicon-chip-density-nears-physical-limit>.)

ПОЛИТИКА И ЮРИДИЧЕСКИЕ АСПЕКТЫ

В зависимости от того, с кем вы разговариваете, вопрос о том, выдержит ли закон проверку временем, может выглядеть по-разному, поскольку у каждого своя точка зрения. Цель книги не в том, чтобы убедить вас принять ту или иную позицию, – она просто излагает преобладающее мнение. Например, можно утверждать, что закон Мура настолько же доказан, как и законы термодинамики. Если глубже изучить классическую физику, можно найти множество несоответствий в ее законах и допущениях. Речь не идет о том, чтобы как-то принизить науку, – просто важно отметить, что всё в науке, включая ее законы, находится в постоянном развитии.

Что касается вопроса о том, перестанет ли существовать закон Мура, то, как правило, законы не прекращают действовать – ученые переформулируют их так, чтобы они лучше соответствовали фактам. Правда? Мы действительно так сказали? Ну, на самом деле, мы имели в виду, что закон Мура может претерпеть такую же трансформацию. Линейные или чрезмерно упрощенные законы редко применимы в общем смысле, поскольку в природе нет прямых линий, в том числе и в ее временных моделях. Поэтому наиболее вероятный сценарий заключается в том, что закон Мура преобразуется в более сигмоидальную функцию, стремясь соответствовать реальности.

Закон Мура напрямую влияет на данные. Все начинается с более умных устройств. Чем умнее устройства, тем шире их распространение. (В наше время электроника повсюду.) Чем шире распространение, тем ниже становится цена, создавая бесконечный цикл, который стимулировал и продолжает стимулировать использование мощных вычислительных машин и маленьких датчиков повсеместно. При наличии большого объема компьютерной памяти и более емких дисков для хранения данных последствием становится расширение доступности данных, таких как веб-сайты, записи транзакций, множество различных измерений, цифровые изображения и другие виды данных, поступающих отовсюду.

Поиск данных повсюду

Ученые начали бороться с впечатляющими объемами данных за много лет до того, как появился термин *большие данные*. В то время интернет еще не производил таких огромных объемов данных, как сегодня. Полезно помнить, что большие данные — это не просто модное явление, созданное производителями программного и аппаратного обеспечения, а основа во многих следующих областях.

» **Астрономия.** Подумайте о данных, получаемых с космических аппаратов в ходе миссий (таких как «Вояджерили «Галилео»), и обо всех данных, получаемых с радиотелескопов — специализированных антенн, используемых для приема радиоволн от астрономических объектов. Типичный пример — проект SETI (Search for Extraterrestrial Intelligence, «Поиск внеземного разума») (<https://www.seti.org>), который ищет внеземные сигналы, наблюдая за радиочастотами, приходящими из космоса. Данные основаны на техносигнатурах — миллионах техносигнатур, которые генерируют огромные объемы данных (<https://www.sciencealert.com/scientists-detected-26-million-possible-technosignatures-they-all-came-from-us>), даже если они в основном поступают из человеческих источников. Недавно действительно был получен сигнал от Проксимы Центавра (<https://theconversation.com/seti-new-signal-excites-alien-hunters-heres-how-we-could-find-out-if-its-real-152498>), но это скорее исключение, чем правило (заставляя всех чувствовать себя как мамы, которым не звонят дети). Другой пример поразительного объема данных, получаемых с различных телескопов, — создание первого изображения черной дыры в центре галактики M87 (подробнее см. на <https://www.jpl.nasa.gov/edu/news/2019/4/19/how-scientists-captured-the-first-image-of-a-black-hole>).

- » **Метеорология.** Представьте попытку предсказать погоду на ближайшее время, учитывая большое количество необходимых измерений, таких как температура, атмосферное давление, влажность, ветры и осадки в разное время, в разных местах и на разных высотах. Прогнозирование погоды действительно одна из первых задач в области больших данных, причем весьма актуальная. По сведениям компании Weather Analytics, предоставляющей климатические данные, более 33% мирового валового внутреннего продукта (ВВП) определяется тем, как погодные условия влияют на сельское хозяйство, рыболовство, туризм и транспорт, и это лишь некоторые примеры. Начиная с 1950-х годов первые суперкомпьютеры того времени использовались для обработки максимально возможного объема данных, поскольку в метеорологии чем больше данных, тем точнее прогноз (подробнее см. на <https://bigdataanalyticsnews.com/big-data-enhance-weather-forecasting>).
- » **Физика.** Рассмотрим огромные объемы данных, производимых экспериментами на ускорителях частиц в попытке определить структуру материи, пространства и времени. Например, Большой адронный коллайдер (<https://home.cern/science/accelerators/large-hadron-collider>), крупнейший из когда-либо созданных ускорителей частиц, ежегодно производит 15 ПБ (петабайт, или 1 миллион гигабайт) данных в результате столкновений частиц (<https://home.cern/science/computing>).
- » **Геномика.** Секвенирование одной цепочки ДНК, то есть определение точного порядка множества комбинаций четырех оснований – аденина, гуанина, цитозина и тимина, составляющих структуру молекулы, требует огромного количества данных. Например, одна хромосома – структура, содержащая ДНК в клетке, – может требовать от 50 до 300 МБ. У человека 46 хромосом, и данные ДНК только одного человека занимают целый DVD-диск. Только представьте, какое огромное хранилище требуется для документирования данных ДНК большого количества людей или для секвенирования других форм жизни на Земле (см. на <https://www.zymergen.com/blog/the-art-of-biological-data-science-bringing-biologys-great-unknowns-into-focus>).
- » **Океанография.** В океанах размещено множество датчиков, измеряющих температуру, течения и другие важные параметры. Кроме того, с помощью гидрофонов можно

записывать звуки для акустического мониторинга как в научных целях (изучение рыб, китов и планктона), так и для целей военной обороны (обнаружение скрытных подводных лодок других стран). Как ни странно, пандемия COVID-19 облегчила сбор еще большего количества данных (подробности можно прочитать по ссылке <https://www.eurekalert.org/news-releases/749064>).

- » **Спутники.** Съемка изображений всего земного шара и их передача на Землю для мониторинга земной поверхности и атмосферы – не новое явление (ТИРОС-1 – первый спутник, передававший изображения и данные, – был запущен еще в 1960 году). Вопрос, сколько именно спутников находится на орбите, довольно сложный, но представление о количестве активных спутников и их назначении можно получить по ссылке <https://www.geospatialworld.net/blogs/how-many-satellites-are-orbiting-the-earth-in-2021>. Объем данных, поступающих на Землю, поражает воображение. Они используются как в военных целях (наблюдение), так и в гражданских – для отслеживания экономического развития, мониторинга сельского хозяйства и наблюдения за изменениями и рисками. Только один спутник Европейского космического агентства, Sentinel-1A, генерирует 5 ПБ данных за два года работы, как можно прочитать на <https://spaceflightnow.com/2016/04/28/europes-sentinel-satellites-generating-huge-big-data-archive>.

Помимо этих традиционных источников данных, новые массивы информации теперь генерируются и передаются через интернет, создавая новые проблемы и требуя решений как в области хранения данных, так и в сфере алгоритмов их обработки:

- » Статистика использования интернета постоянно растет, и невозможно быть уверенным, что цифры, которые вы читаете сегодня, останутся актуальными завтра (<https://www.internetadvisor.com/key-internet-statistics>). В последние годы доступ к интернету получило еще больше людей, причем не только через компьютеры, но и через мобильные устройства, такие как смартфоны. Пандемия COVID-19 только усилила эту тенденцию, поскольку вынудила многих оставаться дома, совершать покупки и работать удаленно.
- » Интернет вещей (IoT) стал реальностью. Вы наверняка слышали этот термин много раз за последние 20 лет, но сейчас количество устройств, подключенных к интернету,

резко возросло. Идея заключается в том, чтобы оснастить всё датчиками и передатчиками и использовать получаемые данные как для лучшего контроля происходящего в мире, так и для создания более «умных» объектов (а если вы приверженец теории заговора, то это еще и удобный способ для правительства следить за людьми). Передающие устройства становятся все меньше, дешевле и экономичнее; некоторые уже настолько малы, что их можно разместить где угодно. IoT-устройства появляются и в неожиданных формах (<https://www.asme.org/topics-resources/content/9-cool-iot-devices-for-our-daily-lives>). По оценкам экспертов, в 2019 году IoT-устройства уже производили около 17,3 ЗБ данных (зеттабайт равен 1 миллиону петабайт). Ожидается, что к 2025 году эта цифра вырастет более чем в четыре раза и достигнет 73,1 ЗБ (<https://dataprot.net/statistics/iot-statistics>).

Внедрение алгоритмов в бизнес

Человечество сейчас находится на беспрецедентном пересечении технологий: огромные объемы данных генерируются все более компактным и мощным оборудованием и анализируются алгоритмами, которые были разработаны благодаря этому же процессу. Дело не только в объеме, который сам по себе представляет собой сложную задачу. Как было формализовано исследовательской компанией Gartner в 2001 году, а затем переосмыслено и расширено другими компаниями, такими как IBM, большие данные можно охарактеризовать четырьмя V, представляющими их ключевые характеристики (<https://edudataonline.com/4-vs-of-big-data>):

- » **Volume (объем)** – количество данных;
- » **Velocity (скорость)** – скорость генерации данных;
- » **Variety (разнообразие)** – количество и типы источников данных;
- » **Veracity (достоверность)** – качество и авторитетность данных (количественная оценка ошибок, некачественных данных и шума, смешанного с сигналами), мера неопределенности данных.

Читайте далее подробности об этих ключевых характеристиках.

Объем

Каждая характеристика больших данных представляет собой как вызов, так и возможность. Например, объем данных создает проблемы для систем хранения, заставляя организации пересматривать существующие методы и решения и подталкивая современные технологии и алгоритмы к поиску новых путей. Однако объем — это еще и возможность, поскольку во всем этом массиве данных можно ожидать определенное количество полезной информации. Эта полезная информация открывает новые возможности и помогает достичь результатов, которые были недостижимы. В качестве примера рассмотрим, как увеличение объема данных может помочь в понимании меняющихся предпочтений потребителей на глобальных рынках или в улучшении обслуживания клиентов компании.

Скорость

К проблемам с объемом добавляется и то, что скорость требует своевременного получения данных, иначе они ускользнут из поля зрения. Для организаций, которые достаточно быстры, чтобы уловить эти данные, скорость открывает возможность реагировать на угрозы в реальном времени и создает дополнительные возможности для получения различных преимуществ (например, финансовой выгоды). Неважно, идет ли речь о данных по угрозам безопасности или финансовым рынкам, — они могут информировать вас обо всем, что происходит под солнцем.

Разнообразие

Разнообразие требует работы с данными во множестве форматов. Некоторые форматы ожидаемы, поскольку вы их спланировали и разработали; другие — неожиданны и сложны, поскольку созданы другими организациями. Разнообразие означает, что вы не можете рассчитывать найти то, что ищете, в определенных местах или в легко понятном виде. Методы поиска и обработки информации в эпоху больших данных становятся гораздо более гибкими, требуя навыков, больше похожих на навыки исследователя или следователя, чем библиотекаря или кладовщика.

Достоверность

Характеристика достоверности способствует демократизации самих данных. В прошлом организации накапливали данные, поскольку они были ценными и труднодоступными. Сейчас различные источники создают данные в таких растущих объемах, что их накопление теряет смысл (90% мировых данных было создано за последние два года), поэтому ограничение доступа бессмысленно. Данные превращаются в такой товар, что по всему миру действует множество программ открытых данных. (У Соединенных Штатов давняя традиция открытого доступа; первые программы открытых данных появились в 1970-х годах, когда Национальное управление океанических и атмосферных исследований [NOAA] начало свободно предоставлять метеорологические данные общественности.) Однако, поскольку данные стали товаром, возникла проблема их достоверности. Вы больше не знаете, насколько данные правдивы, поскольку можете даже не знать их источник.



ЗАПОМНИТЕ

Данные стали настолько вездесущими, что их ценность больше не заключается в самой информации (например, в данных, хранящихся в базе данных компании). Ценность данных заключается в том, как вы их используете. Здесь в игру вступают алгоритмы, меняющие ее правила. Такая компания, как Google, питается свободнодоступными данными, такими как содержимое веб-сайтов или текст, найденный в общедоступных источниках и книгах. Тем не менее ценность, которую Google извлекает из данных, в основном происходит из ее алгоритмов. Например, ценность данных заключена в алгоритме PageRank (рассматриваемом в главе 11), который является основой бизнеса Google. Ценность алгоритмов верна и для других компаний. Система рекомендаций Amazon вносит значительный вклад в доходы компании. Многие финансовые фирмы используют алгоритмическую торговлю и робо-консультинг, используя свободнодоступные данные об акциях и экономическую информацию для инвестиций.

§ 2. Потокотые потоки данных

Когда данные поступают в огромных количествах, их полное хранение может быть затруднительным или даже невозможным. Более того, хранить всё может быть просто бессмысленно. Вот некоторые цифры из статистики за 2019–2020 годы, показывающие, что происходит в интернете всего за одну минуту:

- » 188 миллионов отправленных электронных писем;
- » 350 000 новых твитов в Twitter;
- » 3,8 миллиона поисковых запросов в Google;

Учитывая такие объемы, накапливать данные целый день для последующего анализа может быть неэффективно. Можно просто сохранить их где-нибудь и проанализировать в другой день (это распространенная стратегия архивирования, типичная для баз и хранилищ данных). Однако полезные запросы к данным обычно касаются самых свежих данных в потоке, а с «возрастом» данные становятся менее полезными (в некоторых секторах, например в финансовом, день — это очень много времени). Более того, завтра можно ожидать еще больший объем данных (количество данных растет ежедневно), и это затрудняет, если не делает невозможным, извлечение данных из хранилищ по мере поступления новых. Извлечение старых данных из хранилищ, пока поступают свежие данные, сродни наказанию Сизифа. Согласно греческому мифу, Сизиф получил страшное наказание от бога Зевса: он обречен был вечно катить огромный валун на вершину холма, только чтобы каждый раз смотреть, как тот скатывается обратно (подробнее см. на https://www.mythweb.com/encyc/gallery/sisyphus_c.html).

Иногда данные могут поступать настолько быстро и в таком большом количестве, что записать их на диск просто невозможно: новая информация поступает быстрее, чем требуется времени для записи на жесткий диск. Эта проблема типична для экспериментов с частицами на ускорителях, таких как Большой адронный коллайдер, что вынуждает ученых решать, какие данные сохранять (<https://home.cern/science/computing/storage>). Конечно, можно какое-то время держать данные в очереди, но недолго, поскольку очередь быстро вырастет и станет неуправляемой. Например, если хранить данные очереди в памяти, это вскоре приведет к ошибке нехватки памяти.

Поскольку новые потоки данных могут сделать предыдущую обработку старых данных устаревшей, а откладывание решения проблемы не выход, люди разработали много стратегий для мгновенной обработки массивных и изменчивых объемов данных. Существует три способа работы с большими объемами данных:

- » **хранение.** Некоторые данные хранятся как есть, поскольку могут помочь ответить на неясные вопросы позже. Этот метод опирается на технологии для их немедленного сохранения и последующей быстрой обработки, независимо от того, насколько они массивны;
- » **суммирование.** Некоторые данные обобщаются, поскольку хранить их все в исходном виде нецелесообразно, — сохраняется только важная информация;

» **потребление.** Оставшиеся данные потребляются, поскольку их использование предопределено. Алгоритмы могут мгновенно считывать, обрабатывать и преобразовывать данные в сигналы или действия. После этого система навсегда забывает эти данные.

Первый пункт рассматривается в главе 13, где речь идет о распределении данных между несколькими компьютерами и о понимании алгоритмов, используемых для работы с ними (стратегия «разделяй и властвуй»). Следующие блоки посвящены второму и третьему пунктам применительно к потоковым данным в системах.



СОВЕТ

Когда речь заходит о массивных данных, поступающих в компьютерную систему, часто используют «водные» метафоры: *потоковые данные*, *потоки данных*, «*пожарный шланг*» данных. Работа с потоками данных похожа на использование водопроводной воды: открывая кран, вы можете собирать воду для хранения в чашках или бутылках, использовать ее для приготовления пищи, мытья продуктов, посуды или рук. В любом случае после использования большая часть или вся вода исчезает, но она была очень полезной и действительно жизненно важной.

Анализ потоков с правильным рецептом

Для потоковых данных нужны потоковые алгоритмы, и ключевой нюанс, который нужно знать о потоковых алгоритмах, заключается в том, что, за исключением нескольких показателей, которые они могут вычислить точно, потоковые алгоритмы обязательно дают приближенные результаты. Выходные данные алгоритма почти верны; он не угадывает точный ответ, но выдает близкий к нему.

Помимо «водной» метафоры из предыдущего блока, другая полезная метафора для понимания потоков связана с комбинированием ингредиентов в рецепте. (Мы действительно не пытаемся вас намочить или сделать голодными.) При работе с потоками нужно явно сосредоточиться только на интересующих показателях и опустить многие детали. Вас могут интересовать статистические измерения, такие как среднее значение, минимум или максимум. Более того, вы можете захотеть подсчитать элементы в потоке или отличить старую информацию от новой. Есть много алгоритмов для использования в зависимости от задачи, но в рецептах всегда используются одни и те же ингредиенты. Секрет приготовления идеального потока заключается в использовании в качестве ингредиентов одного или всех алгоритмических инструментов. Среди них:

- » **выборка.** Уменьшите поток до более управляемого объема данных. Представьте весь поток или самые последние наблюдения, используя скользящее окно данных;
- » **хеширование.** Сведите бесконечное разнообразие потока к ограниченному набору простых целых чисел (как описано в параграфе «Опираясь на хеширование главы 7»);
- » **скетчинг.** Позволяет создать краткую сводку необходимых измерений, отбрасывая менее важные детали. Такой подход дает возможность эффективно использовать простое рабочее хранилище – оперативную память компьютера или жесткий диск.

Другая важная характеристика алгоритмов, работающих с потоками данных, — их простота и низкая вычислительная сложность. Потоки данных могут быть очень быстрыми. Алгоритмы, требующие слишком много вычислений, рискуют пропустить важные данные, которые будут безвозвратно утеряны. В таком свете становится понятной польза хеш-функций — они моментально преобразуют входные данные в более удобную для обработки и поиска форму. Хеш-функции работают быстро, поскольку сложность обеих операций составляет $O(1)$ (подробнее о сложности см. в блоке «Работа с функциями» в главе 2). Можно также оценить преимущества методов скетчинга и выборки, которые основаны на идее *сжатия с потерями* (подробнее о сжатии — в главе 14). Сжатие с потерями позволяет представить что-то сложное в более простой форме. Вы теряете некоторые детали, но значительно экономите машинное время и память.

Выборка означает извлечение ограниченного набора примеров из потока и работу с ними как с представителями всего потока. Выборка — хорошо известный инструмент в статистике, позволяющий делать выводы о более широком контексте (технически называемом *генеральной совокупностью*, или *популяцией*) на основе его небольшой части.

Сохраняя нужные данные

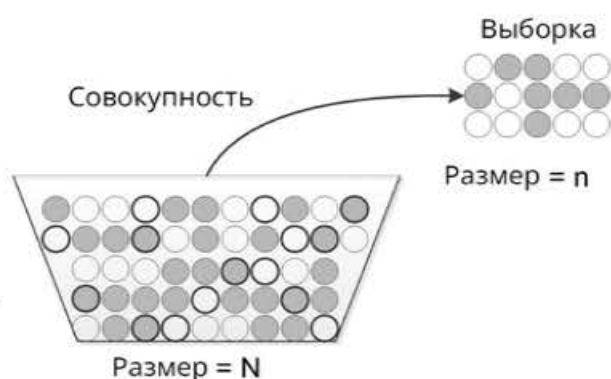
Статистика зародилась в то время, когда проведение полной переписи населения было практически невозможно. *Перепись* — это систематическое исследование населения, включающее его подсчет и сбор полезных данных. Правительство опрашивает всех жителей страны о месте проживания, семье, повседневной жизни и работе. Перепись берет начало в древности. В Библии перепись упоминается в Книге Чисел, где подсчитывается население израильтян после исхода из Египта. В Древнем Риме регулярно проводили перепись населения обширной империи для налогообложения. Исторические документы свидетельствуют о подобных переписях в древних Египте, Греции, Индии и Китае.

Статистика, в частности раздел, называемый *выводной статистикой*, может достичь тех же результатов, что и перепись, с приемлемой погрешностью, с опросом меньшего числа людей (называемое *выборкой*). Таким образом, опросив небольшое количество людей, социологи могут определить общественное мнение более широкой популяции по различным вопросам, например о том, кто победит на выборах. В США, к примеру, статистик Нейт Сильвер прославился тем, что предсказал победителя президентских выборов 2020 года во всех 50 штатах, используя данные выборочных опросов (<https://projects.fivethirtyeight.com/2020-election-forecast>).

Очевидно, что проведение переписи населения влечет огромные затраты (чем больше население, тем выше затраты) и требует серьезной организации (поэтому переписи проводятся нечасто), тогда как статистическая выборка быстрее и дешевле. Сниженные затраты и меньшие организационные требования тоже делают статистику идеальной для потоковой обработки больших данных: пользователям потоковой обработки больших данных не нужна каждая крупинка информации — они могут обобщать сложность данных.

Однако при использовании статистических выборок возникает проблема. В основе статистики лежит выборочный метод, а он требует случайного отбора нескольких примеров из всей совокупности. Ключевой элемент этого подхода заключается в том, что у каждого элемента совокупности должна быть абсолютно одинаковая вероятность попадания в выборку. Если население составляет миллион человек, а размер выборки равен единице, вероятность попадания каждого человека в выборку составляет одну миллионную. В математическом выражении, если обозначить совокупность переменной N , а размер выборки — n , вероятность попадания в выборку составляет n/N , как показано на рисунке 12.2. Представленная выборка — это *простая случайная выборка*. (Существуют и другие типы выборок, более сложные; это простейший тип выборки, на основе которого строятся все остальные.)

РИСУНОК 12.2.
Как работает
выборка из
контейнера



Использование простой случайной выборки похоже на лотерею, но для извлечения нескольких чисел, представляющих целое, все числа должны находиться в контейнере. Потoki данных нелегко поместить в хранилище, из которого можно извлечь выборку; вместо этого приходится извлекать выборку на лету. На самом деле, нужен другой тип выборки, называемый *резервуарной выборкой*. Подобно тому, как водохранилище удерживает воду для последующего использования, хотя вода в нем не стоит на месте, поскольку одна часть прибывает, а другая убывает, этот алгоритм работает путем случайного выбора элементов, которые сохраняются в качестве образцов, пока не появятся другие элементы, чтобы их заменить. Этот алгоритм используется как для мониторинга сетей интернет-коммуникации, так и в системах выполнения запросов в базах данных и поисковых системах, таких как Google. Алгоритм особенно полезен для преобразования больших потоков данных в более управляемые наборы, способные давать приближенную, но надежную информацию.

Алгоритм резервуарной выборки сложнее, чем другая алгоритмическая стратегия, называемая *оконной обработкой*, при которой создается очередь и новым элементам разрешается в нее входить (см. рисунок 12.3). Старые элементы покидают очередь на основе определенного триггера. Этот метод применяется, когда нужны отчеты из потока через точные временные интервалы. Например, может потребоваться узнать, сколько страниц пользователи запрашивают с интернет-сервера каждую минуту. При использовании оконной обработки вы начинаете ставить в очередь запросы страниц в определенную минуту, накапливаете элементы подсчета страниц за конкретный интервал, подсчитываете элементы в очереди, формируете отчет, очищаете очередь и начинаете очередь заново.

Еще одна причина использования оконной обработки — необходимость работать с фиксированным объемом самых последних данных. В этом случае при добавлении каждого нового элемента в очередь самый старый элемент удаляется. Очередь представляет собой структуру «первым пришел — первым ушел» (FIFO), о которой говорилось в главе 6.

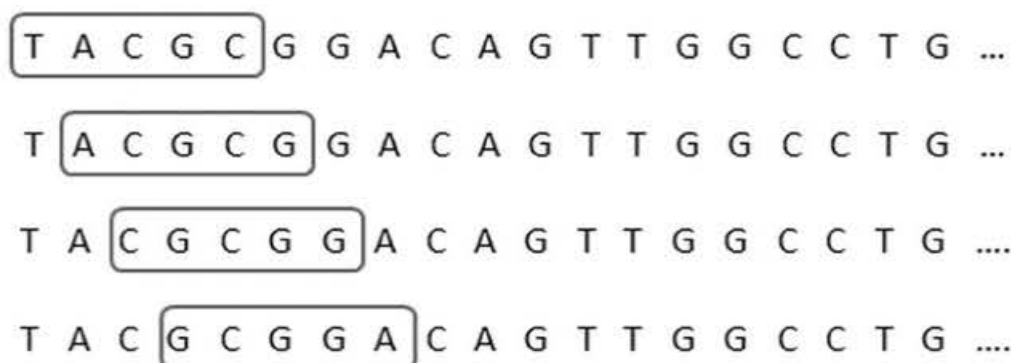


РИСУНОК 12.3.
Пример окон-
ной обработки
потока данных
ДНК

Оконная обработка анализирует выборки с помощью скользящего окна — показывает элементы в пределах окна, которые представляют определенный временной срез или сегмент потока. В отличие от этого, резервуарная выборка охватывает весь поток, предоставляя управляемый объем данных в виде статистической выборки из потока.

Вот как работает резервуарная выборка: при наличии потока данных, содержащего множество элементов, резервуарная выборка инициализируется элементами, взятыми из потока, пока выборка не будет заполнена. Например, если выборка содержит 1000 элементов (это количество обычно помещается во внутреннюю память компьютера), начинают с выбора первых 1000 элементов потока. Количество элементов, которое вы хотите иметь в выборке, обозначается как k , причем k подразумевает выборку, помещающуюся в память компьютера. После того как зарезервированы первые k элементов потока, алгоритм начинает делать свой выбор:

- 1** С начала потока алгоритм подсчитывает каждый новый поступающий элемент. Для отслеживания подсчета используется переменная n . Когда алгоритм начинает работу, значение n равно k .
- 2** По мере поступления новых элементов значение n увеличивается. Новый элемент, поступающий из потока, имеет вероятность k/n быть включенным в резервуарную выборку и вероятность $(1 - k/n)$ не быть включенным.
- 3** Вероятность проверяется для каждого нового поступающего элемента. Это похоже на лотерею: если вероятность подтверждается, новый элемент включается в выборку. В противном случае новый элемент отбрасывается. Если элемент включается, алгоритм удаляет

старый элемент из выборки согласно определенному правилу (самый простой вариант — выбрать старый элемент случайным образом) и заменяет его новым элементом.

Следующий код демонстрирует простой пример на Python, чтобы вы могли увидеть этот алгоритм в действии. В примере используется последовательность букв алфавита (представьте, что это поток данных) и создается выборка из пяти элементов. (Этот код можно найти в загружаемом файле исходного кода **A4D2E; 12: Managing Big Data.ipynb**; подробнее см. во введении.)

```
import string
datastream = list(string.ascii_uppercase)
datastream += list(string.ascii_lowercase)
print(datastream)

['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L',
 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',
 'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v',
 'w', 'x', 'y', 'z']
```

Помимо строк, в примере используются функции из пакета **random** для создания начального значения (для стабильных и воспроизводимых решений) и, при генерации случайного целого числа, проверяется необходимость замены элемента в резервуаре. Кроме значения начального числа, вы можете поэкспериментировать с изменением размера выборки или даже подать алгоритму другой поток (для корректной работы примера он должен быть в виде списка Python).

```
from random import seed, randint
seed(9) # change this value for different results
sample_size = 5
sample = []

for index, element in enumerate(datastream):
    # Until the reservoir is filled, we add elements
    if index < sample_size:
        sample.append(element)
    else:
        # Having filled the reservoir, we test a
        # random replacement based on the elements
        # seen in the data stream
        drawn = randint(0, index)
        # If the drawn number is less or equal the
        # sample size, we replace a previous
```



```

# element with the one arriving from the
# stream
if drawn < sample_size:
    sample[drawn] = element

print(sample)

['y', 'e', 'v', 'F', 'i']

```

Эта процедура гарантирует, что в любое время ваша резервуарная выборка репрезентативна для всего потока данных. В данной реализации переменная `index` играет роль n , а переменная `sample_size` выступает в качестве k . Обратите внимание на две особенности этого алгоритма:

- » По мере роста переменной `index` из-за наплыва данных в поток вероятность попадания в выборку уменьшается. Как следствие, в начале потока многие элементы входят в выборку и покидают ее, но скорость изменений снижается по мере продолжения потока.
- » Если проверить вероятность добавления отдельного элемента из потока данных в выборку и усреднить все значения, среднее будет приближаться к вероятности того, что конкретный элемент из потока данных (генеральной совокупности) будет выбран в любую выборку (необязательно именно в эту), которая равна k/n .

§ 3. Скетчинг ответа на основе потоковых данных

Выборка — отличная стратегия для работы с потоками, но она не отвечает на все вопросы, которые могут возникнуть при анализе потока данных. Например, выборка не может сказать вам, встречался ли уже элемент потока ранее, поскольку она не содержит всю информацию о потоке. То же самое относится к таким задачам, как подсчет количества уникальных элементов в потоке или вычисление частоты появления элементов.

Для достижения таких результатов вам понадобятся хеш-функции (как рассматривалось в главе 7) и скетчи — простые и приближенные сводки данных. В следующих блоках мы начнем с хеш-функций, и вы узнаете, как правильно определять, появлялся ли ранее прибывающий элемент потока, даже если ваш поток бесконечен и вы не можете хранить точную память обо всем, что протекло ранее.

Фильтрация элементов потока по существу

В основе многих потоковых алгоритмов лежат фильтры Блума, созданные почти 50 лет назад Бертоном Х. Блумом, когда компьютерная наука была еще совсем молодой. Первоначальной целью создателя этого алгоритма было найти компромисс между пространством (памятью) и/или временем (сложностью) в обмен на то, что он назвал *допустимыми ошибками*. Его оригинальная статья называется «*Компромиссы пространства/времени в хеш-кодировании с допустимыми ошибками*» (подробнее см. на <https://dl.acm.org/doi/10.1145/362686.362692>).

Вы можете задаться вопросом о пространстве и времени, которые Блум считает мотиваторами для своего алгоритма. Представьте, что вам нужно определить, появлялся ли элемент уже в потоке, используя некоторую ранее созданную или обработанную структуру данных. Это может быть полезно для различных приложений, особенно в кибербезопасности, безопасном просмотре интернета и даже в электронных кошельках (например, при использовании цифровых валют, таких как биткоин). Подробный список можно найти на <https://iq.opengenus.org/applications-of-bloom-filter>.

Поиск чего-либо в потоке подразумевает, что запись и поиск должны быть быстрыми, поэтому хеш-таблица кажется идеальным выбором. *Хеш-таблицы*, как обсуждалось в главе 7, просто требуют добавления элементов, которые вы хотите записать, и их хранения. Извлечение элемента из хеш-таблицы происходит быстро, поскольку она использует легкообрабатываемые значения для представления элемента, а не сам элемент (который может быть довольно сложным). Однако у хранения как самих элементов, так и индекса к ним есть свои ограничения. Если хеш-таблица столкнется с большим количеством элементов, чем она может обработать, например с элементами непрерывного и потенциально бесконечного потока, то возникнут проблемы с памятью.



ЗАПОМНИТЕ

Важное свойство фильтров Блума заключается в том, что ложноположительные результаты возможны, а ложноотрицательные — нет. Например, поток данных может содержать данные мониторинга электростанции в реальном времени. При использовании фильтра Блума по анализу потока данных мы увидели бы, что ожидаемые показания, вероятно, входят в набор допустимых значений, с некоторой допустимой погрешностью. Однако когда в системе происходит сбой, по тому же анализу видно, что показания не входят в набор допустимых значений. Ложноположительные результаты вряд ли вызовут проблемы, но отсутствие ложноотрицательных означает, что все остаются в безопасности. Из-за возможности ложноположительных результатов такие фильтры, как фильтр Блума, являются вероятностными структурами данных — они дают не точный ответ, а вероятностный.



ЗАПОМНИТЕ

Хеши — отдельные записи в хеш-таблице — работают быстро, поскольку действуют как индекс книги. Вы используете *хеш-функцию* для создания хеша; входными данными служит элемент, содержащий сложные данные, а выходными — простое число, которое выступает индексом этого элемента. Хеш-функция детерминирована, поскольку выдает одно и то же число каждый раз, когда вы подаете ей определенные входные данные. Вы используете хеш для поиска нужной вам сложной информации. Фильтры Блума полезны, поскольку представляют собой экономный способ записи следов множества элементов без необходимости хранить их как в хеш-таблице. Они работают простым способом и используют следующие основные компоненты:

- » **битовый вектор** — список из m битовых элементов, где у каждого бита в элементе может быть значение 0 или 1. Чем больше m , тем лучше, хотя существуют способы оптимального определения его размера;
- » **набор хеш-функций**. Каждая хеш-функция представляет собой отдельное значение. Хеш-функции могут быстро обрабатывать данные в качестве индекса списка и создавать равномерно распределенные результаты в списке из m битовых элементов, которые равномерно распределены от минимального до максимального выходного значения хеша.

Добавление элементов в фильтры Блума

Как правило, фильтры Блума создаются фиксированного размера (в недавно разработанных версиях появилась возможность изменять размер фильтра). Работа с ними заключается в добавлении новых элементов в фильтр и их поиске, когда они уже присутствуют. После добавления элемента его невозможно удалить из фильтра (фильтр обладает неизменяемой памятью). При добавлении элемента в битовый вектор некоторые биты устанавливаются в 1, как показано на рисунке 12.4. В данном случае фильтр Блума добавляет X в битовый вектор.

В битовый вектор можно добавить столько элементов, сколько необходимо. Например, на рисунке 12.5 показано, что происходит при добавлении еще одного элемента Y в битовый вектор. Обратите внимание, что бит 7 одинаков как для X , так и для Y . Следовательно, бит 7 представляет собой коллизию между X и Y . Эти коллизии являются источником потенциальных ложноположительных результатов; из-за них алгоритм может сообщить, что элемент уже добавлен в битовый вектор, когда на самом деле это не так. Использование большего битового вектора делает коллизии менее вероятными и улучшает производительность фильтра Блума, но это происходит за счет как пространства, так и времени.

РИСУНОК 12.4.
Добавление
одного элемен-
та в битовый
вектор

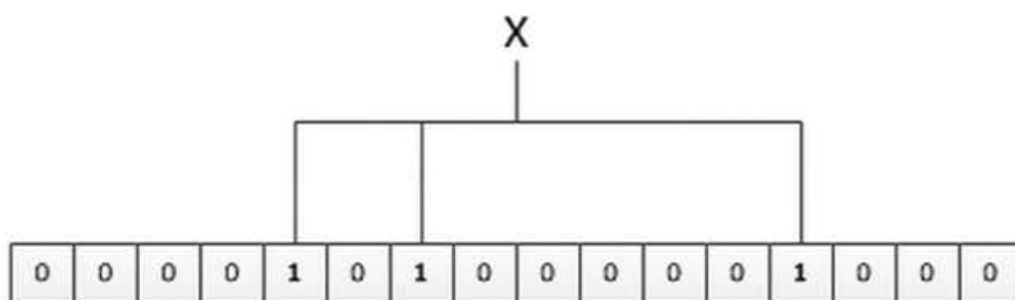
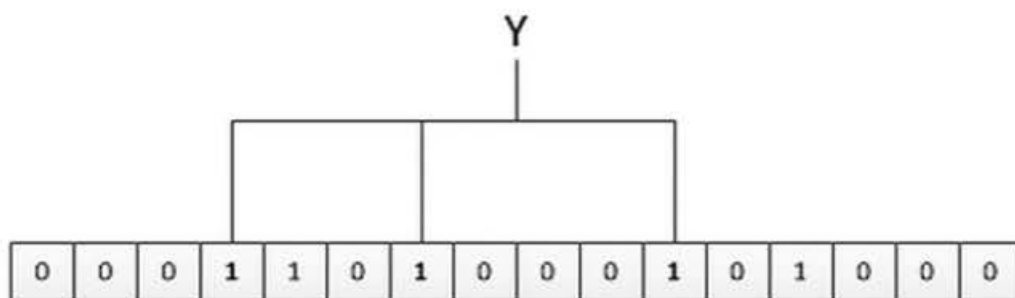


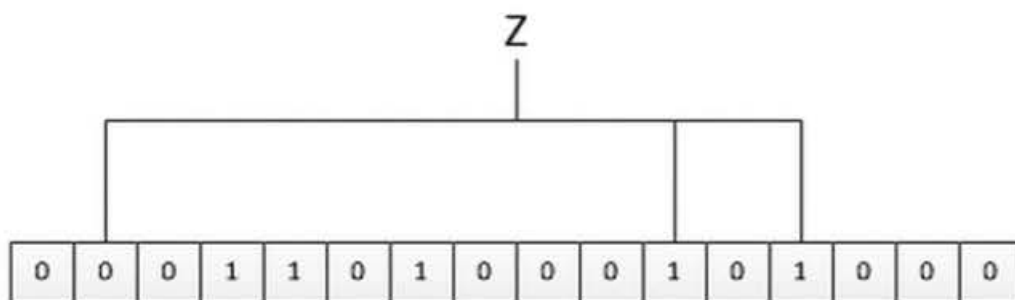
РИСУНОК 12.5.
Добавле-
ние второго
элемента
может вызвать
коллизии



Поиск элемента в фильтре Блума

Поиск в фильтре Блума позволяет определить, присутствует ли конкретный элемент в битовом векторе. В процессе поиска алгоритм ищет наличие 0 в битовом векторе. Например, в предыдущем блоке в битовый вектор были добавлены элементы *X* и *Y*. При поиске элемента *Z* алгоритм находит 0 во втором бите, как показано на рисунке 12.6. Наличие 0 означает, что *Z* не является частью битового вектора.

РИСУНОК 12.6.
Поиск элемента
и определение
его существо-
вания означает
поиск 0 в бито-
вом векторе



Демонстрация фильтра Блума

В этом примере используется Python для демонстрации фильтра Блума с графической визуализацией результатов. Предположим, вы используете поискового робота — специализированное программное обеспечение, которое путешествует по сети, проверяя изменения на отслеживаемых веб-сайтах (что может включать копирование части данных с сайта, известное как *скрапинг*). В примере используется короткий битовый вектор и три хеш-функции — не самая оптимальная конфигурация для обработки большого количества элементов (битовый вектор быстро заполнится), но достаточная для рабочего примера.

```
hash_functions = 3
bit_vector_length = 10
bit_vector = [0] * bit_vector_length

from hashlib import md5, sha1

def hash_f(element, i, length):
    """ This is a magic function """
    h1 = int(md5(element.encode('ascii')).hexdigest(),16)
    h2 = int(sha1(element.encode('ascii')).hexdigest(),16)
    return (h1 + i*h2) % length

def insert_filter(website):
    result = list()
    for hash_number in range(hash_functions):
        position = hash_f(website, hash_number,
                           bit_vector_length)
        result.append(position)
        bit_vector[position] = 1
    print ('Inserted in positions: %s' % result)

def check_filter(website):
    result = list()
    for hash_number in range(hash_functions):
        position = hash_f(website, hash_number,
                           bit_vector_length)
        result.append((position,bit_vector[position]))
    print ('Bytes in positions: %s' % result)
```

Код начинается с создания битового вектора и нескольких функций, которые могут выполнять следующие действия:

- » генерировать множественные хеш-функции (подробнее см. в блоке «Создание собственной хеш-функции в главе 7) на основе алгоритмов хеширования `md5` и `sha1`;
- » вставлять объект в битовый вектор;
- » проверять, включены ли биты, соответствующие объекту в битовом векторе.

Все эти элементы вместе составляют фильтр Блума (хотя битовый вектор — его ключевая часть). В этом примере поисковый робот сначала посещает сайт wikipedia.org, чтобы собрать информацию с нескольких страниц:

```
insert_filter('wikipedia.org')
print(bit_vector)
```

Вот результаты:

```
Inserted in positions: [0, 8, 6]
[1, 0, 0, 0, 0, 0, 1, 0, 1, 0]
```

Эта операция включает биты в позициях 0, 6 и 8 битового вектора. Затем робот обходит сайт youtube.com (где появились новые видео с котятами), и информация об этом посещении вносится в фильтр Блума:

```
insert_filter('youtube.com')
print(bit_vector)
```

Вот обновленные результаты:

```
Inserted in positions: [3, 0, 7]
[1, 0, 0, 1, 0, 0, 1, 1, 1, 0]
```

Здесь фильтр Блума активируется в позициях 0, 3 и 7. С учетом небольшой длины битового вектора уже возникла коллизия в позиции 0, но позиции 3 и 7 — новые. На этом этапе, поскольку алгоритм не может помнить, что он посещал ранее (но посещенные сайты можно проверить с помощью фильтра Блума), пример проверяет, не посещал ли он yahoo.com, чтобы избежать повторных действий, как показано на рисунке 12.7:

```
check_filter('yahoo.com')

Bytes in positions: [(7, 1), (5, 0), (3, 1)]
```

Как графически представлено, в данном случае можно быть уверенным, что пример никогда не посещал yahoo.com, поскольку фильтр Блума сообщает как минимум об одной позиции — позиции 5, бит которой никогда не был установлен.

Вставить Using Hash Function h_1, h_2, h_3

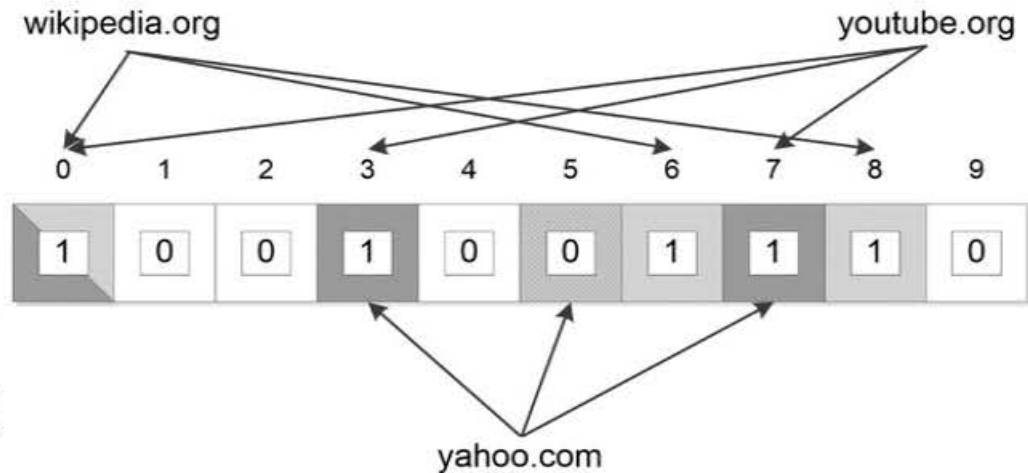


РИСУНОК 12.7.
Проверка при-
надлежности
веб-сайта с по-
мощью фильтра
Блума



СОВЕТ

Поисковый робот обычно занимается получением нового контента с веб-сайтов, избегая копирования уже записанных и переданных данных. Вместо хеширования домена или адреса отдельной страницы можно напрямую заполнять фильтр Блума частью содержимого веб-сайта и использовать его для последующей проверки этого же сайта на наличие изменений.

Существует простой и понятный способ уменьшить вероятность ложноположительных срабатываний. Достаточно увеличить размер битового вектора, который является основой фильтра Блума. Чем больше адресов, тем меньше шансов на коллизии результатов хеш-функций. В идеале размер m битового вектора можно рассчитать, оценив n — ожидаемое количество различных объектов, которые будут добавлены; при этом m должно быть значительно больше n . Оптимальное число k хеш-функций для минимизации коллизий можно затем оценить по следующей формуле ($\ln$ — натуральный логарифм):

$$k = (m/n) \times \ln(2).$$

После определения m , n и k вторая формула помогает оценить вероятность коллизии (частоту ложноположительных срабатываний) при использовании фильтра Блума:

$$\text{false positive rate} = (1 - \exp(-kn/m))^k.$$

Если определить n невозможно из-за разнородности данных в потоке, придется менять либо m — размер битового вектора (что эквивалентно

объему памяти), либо k — количество хеш-функций (что эквивалентно времени), чтобы скорректировать частоту ложноположительных срабатываний. Этот компромисс отражает взаимосвязи пространства, времени и вероятности ошибки, которые Блум рассматривает в своей оригинальной работе.

Определение количества уникальных элементов

Хотя фильтр Блума может отслеживать объекты, поступающие из потока, он не может сказать, сколько их всего. Битовый вектор, заполненный единицами, может (в зависимости от количества хешей и вероятности коллизии) скрывать истинное число объектов, хешируемых по одному адресу.

Знание точного количества различных объектов полезно в разных ситуациях, например когда нужно узнать количество уникальных пользователей, просмотревших определенную веб-страницу, или число различных поисковых запросов. Хранение всех элементов и поиск дубликатов среди них не работают с миллионами элементов, особенно — поступающих из потока. Когда требуется узнать количество различных объектов в потоке, по-прежнему приходится полагаться на хеш-функцию, но подход включает создание числового скетча.

Скетчинг означает получение приближенного значения — неточного, но не полностью неверного ответа. Приближение допустимо, поскольку реальное значение находится недалеко от него. В одном из алгоритмов скетчинга — *HyperLogLog*, основанном на вероятности и аппроксимации, — наблюдают за характеристиками чисел, генерируемых из потока. *HyperLogLog* происходит из исследований компьютерных ученых Найджела Мартина и Филиппа Флажоле. Флажоле усовершенствовал их первоначальный алгоритм *Флажоле — Мартина* (или алгоритм *LogLog*) в более надежную версию *HyperLogLog*, которая работает следующим образом:

- 1 хеш-функция преобразует каждый элемент, полученный из потока, в число;
- 2 алгоритм преобразует число в бинарный формат — базовый числовой стандарт, используемый компьютерами;
- 3 алгоритм подсчитывает количество начальных нулей в бинарном числе и отслеживает максимальное увиденное число, которое обозначается как n ;
- 4 алгоритм оценивает количество уникальных элементов, прошедших в потоке, используя n (количество уникальных элементов равно 2^n).

Например, первый элемент в строке — слово *dog*. Алгоритм хеширует его в целочисленное значение и преобразует в бинарный формат, получая результат 01101010. В начале числа появляется только один ноль, поэтому алгоритм записывает его как максимальное количество начальных нулей. Затем алгоритм встречает слова *parrot* и *wolf*, чьи бинарные эквиваленты — 11101011 и 01101110, оставляя *n* неизменным. Однако когда проходит слово *cat*, результат — 00101110, поэтому *n* становится равным 2. Чтобы оценить количество уникальных элементов, алгоритм вычисляет 2^n , то есть $2^2 = 4$. На рисунке 12.8 показан этот процесс.

СЛОВА	ПРЕОБРАЗОВАНИЕ ХЕША В ДВОИЧНЫЙ КОД	нули
dog →	0 1 1 0 1 0 1 0	= 1
parrot →	1 1 1 0 1 0 1 1	= 0
wolf →	0 1 1 0 1 1 1 0	= 1
cat →	0 0 1 0 1 1 1 0	= 2
		макс = 2

РИСУНОК 12.8.
Подсчет только
начальных
нулей

Хитрость алгоритма заключается в том, что если ваш хеш производит случайные, равномерно распределенные результаты (как в фильтре Блума), то, анализируя бинарное представление, вы можете вычислить вероятность появления последовательности нулей. Поскольку вероятность того, что отдельное бинарное число будет 0, равна одной второй, для расчета вероятности последовательностей нулей вы просто умножаете эту вероятность $\frac{1}{2}$ столько раз, какова длина последовательности нулей:

- » 50% ($\frac{1}{2}$) – вероятность для чисел, начинающихся с 0;
- » 25% ($\frac{1}{2} \times \frac{1}{2}$) – вероятность для чисел, начинающихся с 00;
- » 12,5% ($\frac{1}{2} \times \frac{1}{2} \times \frac{1}{2}$) – вероятность для чисел, начинающихся с 000.
- » $(\frac{1}{2})^k$ – вероятность для чисел, начинающихся с *k* нулей (степени используются для более быстрых вычислений многократного умножения одного и того же числа).



СОВЕТ

Чем меньше чисел видит HyperLogLog, тем больше неточность. Точность повышается при многократном использовании вычислений HyperLogLog с разными хеш-функциями и усреднении результатов каждого вычисления. В качестве альтернативы можно прибегнуть к *стохастическому усреднению*, что означает использование одного и того же хеша, но — разделение потока на группы (например, разделяя элементы на группы по мере их поступления на основе порядка прибытия), и для каждой группы отслеживать максимальное количество конечных нулей. В конце вычисляется оценка уникальных элементов для каждой группы и среднее арифметическое всех оценок.

Учимся подсчитывать объекты в потоке

Последний алгоритм в главе также использует хеш-функции и приближенные скетчи. Он делает это после фильтрации дублированных объектов и подсчета уникальных элементов, появившихся в потоке данных. Умение подсчитывать объекты в потоке данных помогает находить наиболее часто встречающиеся элементы или ранжировать обычные и необычные события. Эта техника применяется для решения таких задач, как поиск самых частых запросов в поисковой системе, самых продаваемых товаров в интернет-магазине, наиболее популярных страниц на веб-сайте или наиболее волатильных акций (путем подсчета количества покупок и продаж акций).

Для решения этой проблемы применяется алгоритм **Count-Min Sketch**, работающий с потоком данных. Он требует всего один проход по данным и хранит минимально возможное количество информации. Этот алгоритм используется во многих реальных ситуациях (например, при анализе сетевого трафика или управлении распределенными потоками данных). Для работы алгоритма требуется набор хеш-функций, каждая из которых связана с вектором счетчиков, что напоминает фильтр Блума, как показано на рисунке 12.9:

- 1 инициализируйте все векторы счетчиков нулевыми значениями во всех позициях;
- 2 при получении объекта из потока примените хеш-функцию для каждого вектора счетчиков, используйте полученный числовой адрес для увеличения значения в соответствующей позиции;
- 3 чтобы оценить частоту появления объекта, примените к нему хеш-функцию и получите значение в соответствующей позиции; из всех значений, полученных из векторов счетчиков, выберите наименьшее — это и будет количество появлений объекта в потоке.

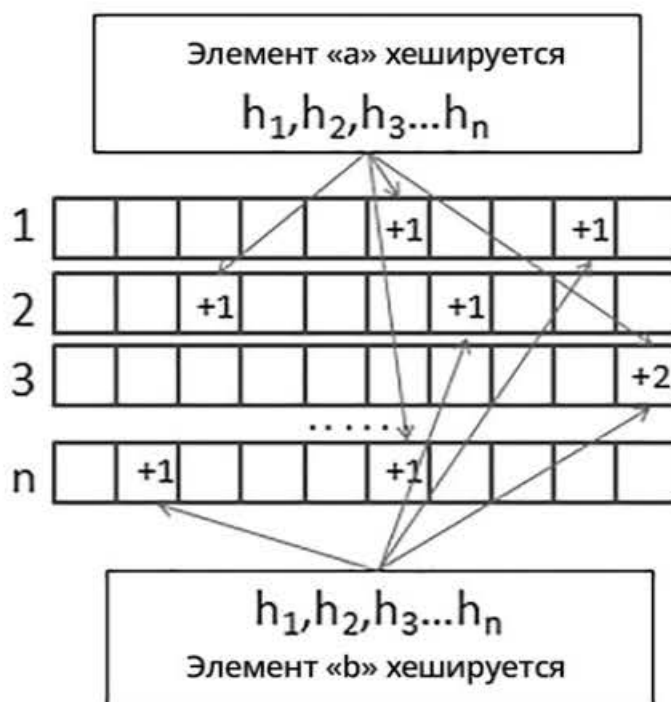


РИСУНОК 12.9.
Обновление
значений
в Count-Min
Sketch



Поскольку коллизии всегда возможны при использовании хеш-функции, особенно если у связанного вектора счетчиков мало слотов, наличие нескольких векторов счетчиков повышает вероятность того, что хотя бы один из них сохранит правильное значение. Следует выбирать наименьшее значение, так как оно меньше всего смешано с ложноположительными подсчетами из-за коллизий.

В ЭТОЙ ГЛАВЕ

- » Почему просто «больше, крупнее и быстрее» – это не всегда правильное решение
- » Рассмотрение подходов к хранению и обработке данных, применяемых интернет-компаниями
- » Понимание того, как использование кластеров дешевого оборудования снижает затраты
- » Преобразование сложных алгоритмов в разделяемые параллельные операции с помощью MapReduce

ГЛАВА 13

Распараллеливание операций

Управление огромными объемами данных с использованием потоковых стратегий или выборки обладает явными преимуществами (как обсуждалось в главе 12), когда нужно иметь дело с массивной обработкой данных. Использование потоковых алгоритмов и алгоритмов выборки помогает получить результат даже при ограниченной вычислительной мощности (например, при использовании собственного компьютера). Однако у этих подходов есть свои издержки.

- » **Потоковая обработка:** справляется с бесконечными объемами данных. Однако алгоритмы работают медленно, поскольку обрабатывают отдельные фрагменты данных, а скорость потока определяет темп работы.
- » **Выборка:** позволяет применять любые алгоритмы на любой машине. Однако полученный результат неточен, поскольку у вас есть только вероятность, а не уверенность в получении правильного ответа. Чаще всего вы получаете лишь правдоподобный результат.

В первом параграфе главы рассматриваются задачи, требующие обработки больших объемов данных как точно, так и своевременно, а также способы избежать затрат на потоковую обработку и выборку. В цифровом мире — множество примеров: поиск по ключевым словам среди миллиардов веб-сайтов или обработка различной информации (поиск изображения в видеохранилище или совпадений во множестве последовательностей ДНК). Последовательное выполнение таких вычислений заняло бы целую жизнь.

Основное решение проблемы сочетания скорости и точности — *распределенные вычисления*, как объясняется в блоке «Распределение файлов и операций». Это означает объединение множества компьютеров в сеть и совместное использование их вычислительных возможностей в сочетании с алгоритмами, работающими на них независимо и параллельно. Здесь в игру вступает второй параграф главы. После представления MapReduce в блоке «Применение решения MapReduce» в главе подробно показано, как настроить обработку MapReduce.



ЗАПОМНИТЕ

Вам не нужно вручную набирать исходный код для этой главы. На самом деле, использовать загружаемый исходный код намного проще. Исходный код главы можно найти в файле `\A4D2E\A4D2E; 13; Map_Reduce.ipynb` из загружаемых материалов. Подробнее о том, как найти эти исходные файлы, смотрите во введении.

§ 1. Управление огромными объемами данных

Использование интернета для выполнения широкого спектра задач, наряду с ростом популярности его наиболее успешных приложений, таких как поисковые системы или социальные сети, заставило профессионалов во многих областях пересмотреть подходы к применению алгоритмов и программных решений для работы с потоком данных. Поиск тем и соединение людей движут этой революцией.

Только представьте прогресс с точки зрения доступных веб-сайтов и страниц за последние 15 лет. Даже если использовать умный алгоритм вроде PageRank (рассмотренный в главе 11), справляться с постоянно растущими и изменяющимися данными все равно сложно. То же самое касается услуг социальных сетей. По мере роста числа пользователей и развития их взаимоотношений лежащий в основе граф, соединяющий их, становится колоссальным по масштабу. При таком масштабе обработка узлов и связей для поиска групп и соединений становится невероятно сложной. (Раздел 3 книги подробно рассматривает графы.)

Помимо коммуникационных данных, рассмотрим онлайн-ритейлеров, предоставляющих виртуальные склады с тысячами и тысячами товаров и услуг (книги, фильмы, игры и т.д.). Хотя вы понимаете, почему купили что-то, ритейлер видит товары в вашей корзине как маленькие части головоломки принятия решения о покупке — головоломки, которую нужно решить, чтобы понять покупательские предпочтения. Решение этой головоломки позволяет ритейлеру предлагать и продавать альтернативные или дополнительные товары.

Понимание параллельной парадигмы

Производители процессоров нашли простое решение, когда столкнулись с задачей втиснуть больше вычислительной мощности в микропроцессоры (как прогнозировал и частично предписывал закон Мура, рассмотренный в главе 12). Однако «больше, крупнее и быстрее» не всегда правильное решение. Когда они обнаружили, что энергопотребление и тепловыделение ограничивают добавление большего количества процессоров на один чип, инженеры пошли на компромисс, создав *многоядерные процессоры* — ЦП, сделанные путем объединения двух или более процессоров вместе. Использование многоядерной технологии сделало параллельные вычисления доступными для более широкой аудитории.

Параллельные вычисления существуют уже давно, но изначально они применялись в основном в высокопроизводительных компьютерах, таких как суперкомпьютеры Cray, созданные Сеймуром Креем в компании Control Data Corporation (CDC) в 1960-х годах. Если говорить простым языком, ассоциативные и коммутативные свойства в математике выражают основную идею параллелизма. Например, при сложении чисел вы можете группировать части сумм вместе или складывать части в порядке, отличном от того, который показан в формулах.

Associative property

$$2 + (3 + 4) = (2 + 3) + 4$$

Commutative property

$$2 + 3 + 4 = 4 + 3 + 2$$

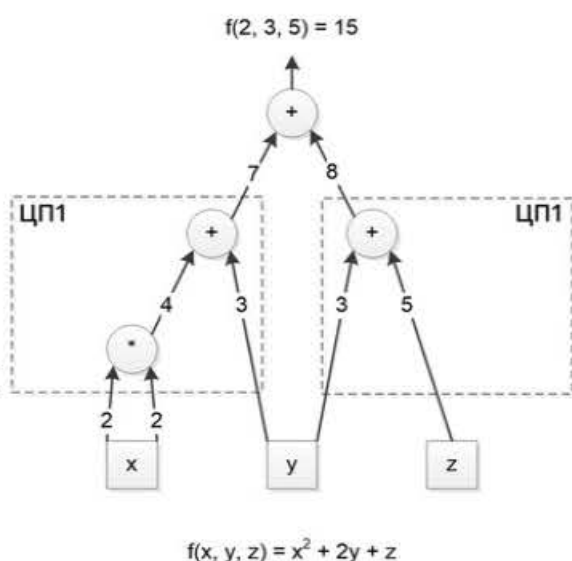
Те же концепции применимы к вычислительным алгоритмам, независимо от того, имеете ли вы дело с последовательностью операций или математической функцией. В большинстве случаев алгоритм можно свести к более простой форме, применяя ассоциативные и коммутативные свойства, как показано на рисунке 13.1. Затем можно разделить части и позволить разным устройствам выполнять атомарные операции независимо, суммируя результат в конце.

В этом примере два процессора разделяют простую функцию с тремя входными параметрами (x , y и z), используя как ассоциативные, так и коммутативные свойства. Для решения уравнения требуется совместное использование общих данных (ЦП1 нужны значения x и y ; ЦП2 нужны значения y и z). Обработка выполняется параллельно до тех пор, пока оба процессора не выдадут свои результаты, которые затем суммируются для получения ответа.

Параллелизм позволяет одновременно выполнять большое количество вычислений. Чем больше процессов, тем выше скорость выполнения вычислений, хотя затраченное время не находится в линейной зависимости от количества параллельных исполнителей. (Неверно полагать, что два процессора означают двукратное увеличение скорости, три процессора — трехкратное и т.д.) На самом деле, нельзя ожидать, что ассоциативные или коммутативные свойства будут работать для каждой части вашего алгоритма или компьютерных инструкций. Некоторые части алгоритма просто невозможно распараллелить, как утверждается в законе Амдала, который помогает определить преимущество в скорости за счет параллелизма ваших вычислений (подробнее см. на <https://www.geeksforgeeks.org/computer-organization-amdahls-law-and-its-proof>). Кроме того, другие аспекты могут ослабить положительный эффект параллелизма:

- » **накладные расходы.** Вы не можете суммировать результаты параллельно;
- » **организационные вопросы.** Преобразование из человеко-читаемого языка в машинный требует времени. Обеспечение совместной работы процессоров увеличивает затраты на преобразование, делая невозможным достижение двукратного эффекта от двух процессоров, даже если каждую часть задачи можно выполнить параллельно;
- » **асинхронные выходы.** Поскольку параллельные исполнители не выполняют задачи с одинаковой скоростью, общая скорость ограничивается самым медленным из них. (Как в случае с флотом, где скорость всего флота определяется скоростью самого медленного судна.)

РИСУНОК 13.1.
Ассоциативные
и коммутатив-
ные свойства
позволяют
реализовать
параллелизм



Хотя параллелизм не всегда оказывается настолько эффективным, как ожидалось, он потенциально способен решить проблему обработки огромного количества операций быстрее, чем при использовании одного процессора (если большое число исполнителей может обрабатывать их параллельно). Однако параллелизм не может справиться с массивными объемами данных, лежащими в основе вычислений, без другого решения: распределенных вычислений на распределенных системах.



ЗАПОМНИТЕ

Когда вы покупаете новый компьютер, продавец, вероятно, рассказывает вам о ядрах и потоках. *Ядра* — это процессоры, которые объединены в единый процессорный чип и работают параллельно, используя многопроцессорную обработку. Поскольку каждое ядро независимо, задачи выполняются одновременно. *Потоки*, в свою очередь, относятся к способности одного ядра разделять свою активность между несколькими процессами почти параллельным образом. Однако в этом случае каждый поток поочередно использует процессор, поэтому задачи не выполняются одновременно. Это называется *многопоточностью*.

Распределение файлов и операций

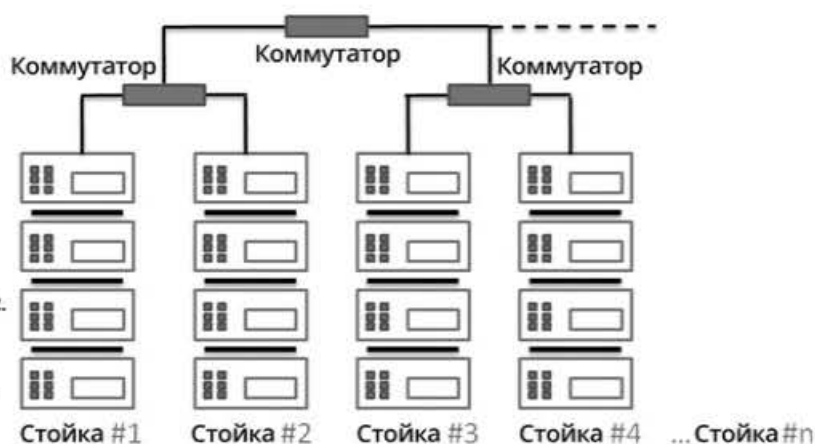
Большие графы, огромные массивы текстовых файлов, изображений и видео, а также гигантские матрицы отношений смежности требуют параллельного подхода. К счастью, вам больше не нужен суперкомпьютер для работы с ними — вместо этого можно положиться на параллелизм, обеспечиваемый группой гораздо менее мощных компьютеров. Тот факт, что эти большие источники данных продолжают расти, означает, что вам

нужен иной подход, чем использование одного компьютера, специально разработанного для их обработки. Данные растут настолько быстро, что к тому времени, когда вы закончите проектирование и производство суперкомпьютера для обработки данных, он может оказаться уже непригодным, поскольку данные успели вырасти слишком сильно.

Все началось в 2004 году, когда концепция MapReduce стала общедоступной благодаря статье «MapReduce: Simplified Data Processing On Large Clusters», написанной Джеффри Дином и Санджаем Гемаватом (доступна по адресу <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.324.78&rep=rep1&type=pdf>). В статье описывается, как Google собирала и анализировала данные веб-сайтов для оптимальной работы своей поисковой системы. Описание авторами подхода MapReduce и его работы в файловой системе Google (GFS) вызвало большой интерес у сообщества разработчиков открытого программного обеспечения, и фонд Apache Software Foundation запустил несколько проектов. Одним из таких проектов стал Nutch (<http://nutch.apache.org>) — поисковая система с открытым исходным кодом; другим — Apache Lucene (<https://lucene.apache.org>), использовавшийся для обеспечения текстового поиска в базах данных. Ситуация стала еще интереснее в 2006 году, когда Даг Каттинг, сотрудник Yahoo!, разработал Hadoop и назвал его в честь игрушечного слона своего сына. Выпущенный как проект с открытым исходным кодом в 2007 году, он оказался в центре внимания и стал проектом высшего уровня в Apache Software Foundation. Те же базовые идеи, лежащие в основе Hadoop, вдохновили и многие другие параллельные аналогичные проекты для обработки больших данных, среди которых наиболее известен Spark (<https://spark.apache.org>) — исследовательский проект лаборатории AMPLab Калифорнийского университета в Беркли, позже переданный Apache Software Foundation.

Большинство этих проектов по обработке больших данных работают схожим образом. Инженеры объединяют множество существующих технологических идей и создают некий вариант распределенной файловой системы (DFS). При использовании DFS данные хранятся не на одном мощном компьютере с гигантским жестким диском; вместо этого DFS распределяет их между несколькими менее мощными компьютерами, сравнимыми с персональным компьютером. Инженеры организуют компьютеры в *кластер* — физическую систему стоек и кабельных соединений. Стойки — настоящая основа сети, в них размещаются компьютеры один рядом с другим. В одной сетевой стойке может находиться различное количество компьютеров — от 8 до 64, каждый из которых соединен с остальными. Стойки соединяются между собой посредством кабельной сети, созданной путем подключения стоек не напрямую друг к другу, а к различным уровням *коммутаторов* — сетевых устройств, способных эффективно обрабатывать и управлять обменом данными между стойками, как показано на рисунке 13.2.

РИСУНОК 13.2.
Схема, представляющая
вычислительный кластер



Все это оборудование можно найти в любом компьютерном магазине, и именно оно делает инфраструктуру DFS жизнеспособной. Теоретически в сеть можно объединить миллион и более компьютеров. (О версии такой системы от Google, согласно последним доступным оценкам, можно прочесть по адресу <https://www.datacenterknowledge.com/archives/2017/03/16/google-data-center-faq>). Интересно, что эти сервисы наращивают вычислительную мощность при необходимости путем добавления новых компьютеров, а не создания новых сетей.

В этой системе при поступлении данных DFS разбивает их на фрагменты (каждый — размером до 64 МБ). DFS создает несколько копий фрагментов и распределяет каждую копию по компьютерам в сети. Процесс разделения данных на фрагменты, их дублирования и распределения происходит довольно быстро, независимо от структуры данных (будь то упорядоченная информация или беспорядочный набор). Единственное требование касается записи адресов фрагментов в DFS, что достигается с помощью индекса для каждого файла (который также реплицируется и распределяется), называемого *главным узлом*. Высокая скорость работы DFS обусловлена способом обработки данных. В отличие от предыдущих методов хранения (таких, как хранилища данных), DFS не требует какой-либо особой сортировки, упорядочивания или очистки самих данных. Напротив, она:

- » обрабатывает данные любого размера, разбивая их на управляемые фрагменты;
- » сохраняет новые данные, добавляя их к старым (DFS никогда не обновляет ранее полученные данные);
- » избыточно реплицирует данные, поэтому не требуется их резервное копирование (дублирование само по себе резервная копия).



ЗАПОМНИТЕ

Компьютеры выходят из строя по разным причинам: из-за жесткого диска, процессора, системы питания или какого-либо другого компонента. Статистически можно ожидать, что компьютер в сети проработает около 1000 дней (примерно три года). (Точное среднее время между отказами, или MTBF, довольно сложно рассчитать из-за большого количества переменных, но статья по адресу <https://www.trentonsystems.com/blog/how-to-calculate-a-rackmount-computer-failure-rate> дает представление о том, что в это входит.) Следовательно, в службе с миллионом компьютеров можно ожидать, что 1000 компьютеров будут выходить из строя каждый день. Именно поэтому DFS распределяет три или более копии ваших данных между несколькими компьютерами в сети. Репликация снижает вероятность потери данных из-за сбоя. Вероятность сбоя, затрагивающего только те компьютеры, где хранится один и тот же фрагмент данных, составляет примерно один из миллиарда (при условии, что DFS реплицирует данные трижды), что делает этот риск ничтожным и приемлемым.

Применение решения MapReduce

Хотя распределенные системы быстро сохраняют данные, их извлечение происходит гораздо медленнее, особенно при выполнении анализа и применении алгоритмов. Та же проблема возникает, когда вы разбиваете пазл на части и разбрасываете их (это легко). Затем нужно собрать части и воссоздать исходное изображение (это сложно и требует времени). При работе с данными в DFS:

- 1 получите главный узел и прочитайте его, чтобы определить местоположение частей файла;
- 2 отправьте запрос на получение ранее сохраненных фрагментов данных компьютерам в сети;
- 3 соберите фрагменты данных, хранящиеся на нескольких компьютерах, на одном компьютере (если это возможно; некоторые файлы могут быть слишком большими для хранения на одной машине).

Очевидно, этот процесс может стать сложным, поэтому инженеры веб-служб решили, что лучше не восстанавливать файлы перед их обработкой. Более разумное решение — оставить их в виде фрагментов на исходных компьютерах и позволить хост-компьютеру их обрабатывать. Только сокращенная версия, которая уже почти полностью обработана, должна будет передаваться по сети, ограничивая передачу данных. MapReduce — это решение, которое предоставляет средства для параллельной обработки алгоритмов в распределенной системе данных. Как алгоритм, MapReduce состоит всего из двух частей: `map` и `reduce`.

ИСПОЛЬЗОВАНИЕ ГОТОВОГО РЕШЕНИЯ ДЛЯ MAPREDUCE

Хотя в книге показано, как создать решение MapReduce с нуля, вам совсем не обязательно изобретать велосипед каждый раз, когда возникает подобная задача. Существуют пакеты, такие как MrJob (название означает «Map Reduce Job»: <https://mrjob.readthedocs.io/en/latest/index.html>), которые позволяют быстро и легко тестировать и выполнять задачи MapReduce прямо на вашем локальном компьютере. Кроме того, с помощью такого пакета, как MrJob, вы можете легко передавать и запускать задачу, используя облачные ресурсы, например Amazon Web Services с Elastic MapReduce (EMR) (<https://aws.amazon.com/emr>) или Hadoop (<http://hadoop.apache.org>). Суть здесь в том, что важно понимать, как работает сам алгоритм (в этом и состоит задача книги), но в некоторых случаях писать весь необходимый код вручную оказывается просто не нужно.

Объяснение `map`

Первая фаза алгоритма MapReduce — это часть `map`, функция, встречающаяся во многих *функциональных языках программирования* (стиль программирования, который рассматривает вычисления как математическую функцию). Функция `map()` проста: вы начинаете с одномерного массива (который в Python может быть списком) и с функции. Применяя функцию к каждому элементу массива, вы получаете массив идентичной формы, значения которого преобразованы. В следующем примере содержится список из десяти чисел, которые функция преобразует в их степенные эквиваленты:

```
def mapper(fun, *iter):
    for i in zip(*iter):
        yield fun(*i)

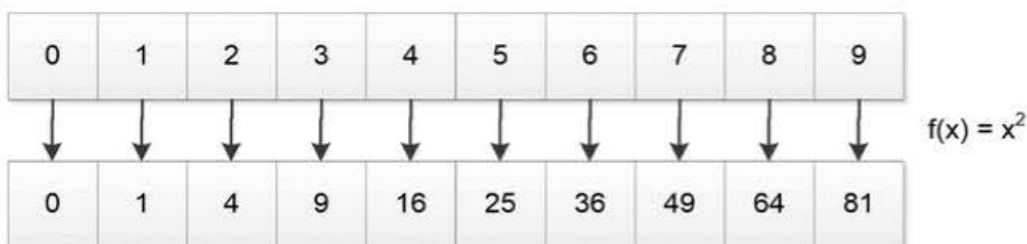
L = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
m = list(mapper(lambda x: x**2, L))
print(m)
```

Вывод показывает, что каждый из результатов — это квадрат входного значения:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```


Функция `mapper()` применяет лямбда-функцию Python. *Лямбда-функция* — это функция, определяемая на лету (см. книгу «Функциональное программирование для чайников» Джона Пола Мюллера [издательство Wiley] для понимания лямбда-функций). В предыдущем примере лямбда-функция преобразует каждый элемент в исходном списке в результирующий элемент. На рисунке 13.3 показан результат этого процесса отображения.

РИСУНОК 13.3.
Отображение
списка чисел
с помощью
функции возве-
дения в квадрат



Обратите внимание, что преобразование каждого элемента списка не зависит от других. Вы можете применять функцию к элементам списка в любом порядке. (Однако результат нужно сохранить в правильной позиции в итоговом массиве.) Возможность обрабатывать элементы списка в любом порядке создает сценарий, который естественным образом распараллеливается без особых усилий.



ЗАПОМНИТЕ

Не все задачи поддаются естественному распараллеливанию, а некоторые никогда не будут параллельными. Тем не менее иногда можно переосмыслить или переформулировать задачу так, чтобы получить набор вычислений, которые компьютер сможет обрабатывать параллельно.

Объяснение **reduce**

Вторая фаза алгоритма MapReduce — это этап **reduce** (есть также промежуточный этап — перемешивание и сортировка, — который будет объяснен в следующем блоке, но пока он неважен, если только вам не хочется потанцевать). Работая со списком, **reduce** применяет функцию последовательно, накапливая результаты. Так, при использовании функции суммирования **reduce** применяет суммирование ко всем элементам входного списка. Функция **reduce** берет первые два элемента массива и объединяет их, затем объединяет этот промежуточный результат со следующим элементом массива — и так далее, пока не обработает весь массив.



СОВЕТ

Вы также можете задать начальное число. Когда вы указываете начальное число, `reduce` начинает с объединения этого числа с первым элементом списка, чтобы получить первый промежуточный результат. В следующем примере используется результат из фазы отображения, который сворачивается с помощью функции суммирования (как показано на рисунке 13.4):

```
def reducer(fun, seq):
    if len(seq)==1:
        return seq[0]
    else:
        return fun(reducer(fun, seq[:-1]), seq[-1])

L = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
m = list(map(lambda x: x**2, L))
r = reducer(lambda x, y: x+y, m)
print(r)
```

```
def reducer(fun, seq):
    if len(seq)==1:
        return seq[0]
    else:
        return fun(reducer(fun, seq[:-1]), seq[-1])
```

Результатом является сумма предыдущих возведенных в квадрат чисел, как показано здесь:

285

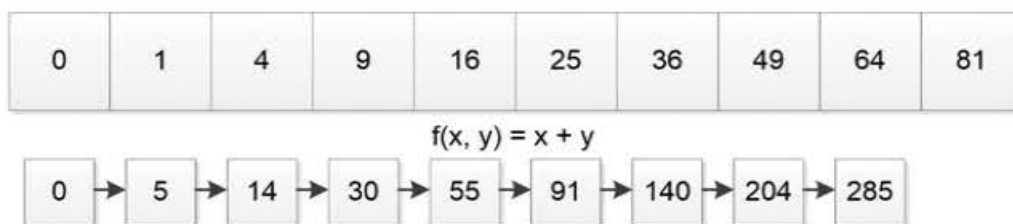


РИСУНОК 13.4.
Сворачивание
списка чисел
в их сумму



ЗАПОМНИТЕ

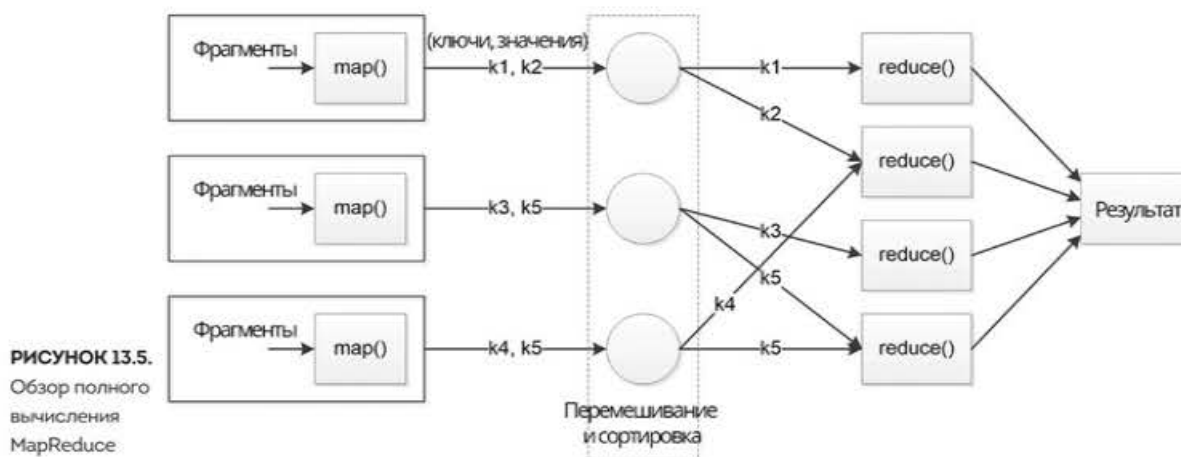
Функция `reducer()` работает с входным массивом как с потоком данных (как обсуждалось в главе 12). Обычно она обрабатывает по одному элементу за раз и отслеживает промежуточные результаты.

Распределение операций

Между фазами `map()` и `reduce()` есть промежуточная фаза — перемешивание и сортировка. Как только задача `map` завершается, хост перенаправляет полученные кортежи пар «ключ — значение» на нужный компьютер в сети для применения фазы `reduce()`. Обычно это делается путем группировки совпадающих пар ключей в единый список и применения хеш-функции к ключу способом, аналогичным фильтрам Блума (см. главу 12). Результатом является адрес в вычислительном кластере для передачи списков.

На другом конце передачи компьютер, выполняющий фазу `reduce()`, начинает получать списки кортежей с одним или несколькими ключами. Множественные ключи возникают при коллизии хешей, которая происходит, когда разные ключи дают одинаковое хеш-значение, и поэтому они попадают на один и тот же компьютер. Компьютер, выполняющий фазу `reduce`, сортирует их в списки, содержащие одинаковые ключи, прежде чем передать каждый список в фазу `reduce`, как показано на рисунке 13.5.

Как показано на рисунке, MapReduce принимает множество входных данных на каждом компьютере вычислительного кластера, где они хранятся, отображает данные и преобразует их в кортежи пар «ключ — значение». Организованные в списки, эти кортежи передаются хостом по сети другим компьютерам, где принимающие компьютеры выполняют операции сортировки и свертки, приводящие к результату.



§ 2. Разработка алгоритмов для MapReduce

В отличие от других примеров в книге, MapReduce можно рассматривать скорее как стиль вычислений или фреймворк для работы с большими данными, чем как алгоритм. Как фреймворк, он позволяет комбинировать различные распределенные алгоритмы (параллельные алгоритмы, распределяющие вычисления между разными компьютерами) и обеспечивает их эффективную и успешную работу с большими объемами данных. Алгоритмы MapReduce можно найти во многих приложениях; подробнее о них можно прочитать в вики Apache, посвященной Hadoop, где описаны компании, использующие его, способы применения и типы вычислительных кластеров: <https://cwiki.apache.org/confluence/display/HADOOP2/PoweredBy>. Хотя возможностей много, чаще всего MapReduce используется для выполнения следующих задач:

- » текстовые алгоритмы для разбиения текста на элементы (токены), создания индексов и поиска релевантных слов и фраз;
- » создание графов и графовые алгоритмы;
- » интеллектуальный анализ данных и обучение новых алгоритмов на основе данных (машинное обучение).

Одно из самых распространенных применений алгоритма MapReduce — обработка текста. Пример в этом блоке демонстрирует, как решить простую задачу подсчета определенных слов в текстовом фрагменте с помощью подхода отображения и свертки, используя многопоточность или многопроцессорность (в зависимости от операционной системы, установленной на вашем компьютере).



ПРИМЕЧАНИЕ

Язык программирования Python не идеален для параллельных операций. Технически, из-за проблем синхронизации и доступа к общей памяти, интерпретатор Python не является потокобезопасным, что означает возможность возникновения ошибок при выполнении приложений, использующих несколько процессов или потоков на нескольких ядрах. Следовательно, Python ограничивает многопоточность одним потоком (код распределяется, но увеличения производительности не происходит), а многоядерный параллелизм с помощью нескольких процессов действительно сложно реализовать, особенно на компьютерах под управлением Windows. Узнать больше о различиях между потоками и процессами можно в статье Microsoft по адресу <https://docs.microsoft.com/windows/win32/procthread/about-processes-and-threads>.

Настройка симуляции MapReduce

Этот пример обрабатывает текст, находящийся в общественном достоянии, полученный с сайта некоммерческой организации Project Gutenberg (<https://www.gutenberg.org>). Первый обрабатываемый текст — роман «Война и мир» Льва Толстого. Следующий код загружает данные в память:

```
from urllib import request

url = 'https://github.com/lmassaron/datasets/releases/'
url += 'download/1.0/2600.txt'
response = request.urlopen(url)
text = response.read().decode('utf-8')[627:]
from urllib import request

url = 'https://github.com/lmassaron/datasets/releases/'
url += 'download/1.0/2600.txt'
response = request.urlopen(url)
text = response.read().decode('utf-8')[627:]

print (text[:37])
```

Вот что вы увидите:

```
WAR AND PEACE

By Leo Tolstoy/Tolstoi
```

Наберитесь терпения! Загрузка книги занимает время (попробуйте прочитать ее так же быстро, как это делает компьютер). После завершения код отображает первые несколько строк с названием. Данные сохраняются в переменной `text`. В рамках процесса текст разбивается на слова и сохраняется в списке, как показано в следующем коде:

```
words = text.split()
print ('Number of words: %i' % len(words))
```

Вывод показывает, что в тексте много слов (пальцы Толстого, должно быть, сильно устали):

```
Number of words: 566218
```

Теперь переменная `words` содержит отдельные слова из книги. Пора импортировать необходимые пакеты Python и функции для примера, используя следующий код:

```

import os
if os.name == "nt":
    #Safer multithreading on Windows
    from multiprocessing.dummy import Pool
else:
    #Multiprocessing on Linux,Mac
    from multiprocessing import Pool
from multiprocessing import cpu_count
from functools import partial
import string

```

В зависимости от вашей операционной системы пример использует многопроцессорную обработку или многопоточность. Windows использует многопоточность, которая разделяет задачу на несколько потоков, обрабатываемых одновременно одним ядром. На системах Linux и Mac код выполняется параллельно и каждая операция обрабатывается отдельным ядром компьютера.

Следующий код подсчитывает слова из списка, соответствующего набору ключевых слов. После удаления знаков препинания код сравнивает слова, и, если он находит совпадение с ключевым словом, функция возвращает кортеж, состоящий из ключа, совпавшего ключевого слова и единичного значения, которое является счетчиком. Этот вывод представляет собой ядро MapReduce `map`:

```

def remove_punctuation(text):
    return ''.join([l for l in text if l in
                    string.ascii_letters])

def count_words(list_of_words, keywords):
    results = list()
    for word in list_of_words:
        for keyword in keywords:
            if keyword == remove_punctuation(
                            word.upper()):
                results.append((keyword,1))
    return results

```

Следующие функции разделяют данные. Этот подход похож на то, как распределенная система разделяет данные. Код распределяет вычисления и собирает результаты:

```
def Partition(data, size):
    return [data[x:x+size] for x in range(0, len(data),
                                          size)]

def Distribute(function, data, cores):
    pool = Pool(cores)
    results = pool.map(function, data)
    pool.close()
    return results
```

Наконец, следующие функции перемешивают и упорядочивают данные для сокращения результатов. Этот шаг представляет собой последние две фазы задачи MapReduce:

```
def Shuffle_Sort(L):
    # Shuffle
    Mapping = dict()
    for sublist in L:
        for key_pair in sublist:
            key, value = key_pair
            if key in Mapping:
                Mapping[key].append(key_pair)
            else:
                Mapping[key] = [key_pair]
    return [Mapping[key] for key in Mapping]

def Reduce(Mapping):
    return (Mapping[0][0], sum([value for (key, value)
                               in Mapping]))
```

Исследование с помощью отображения

Следующий код симулирует распределенную среду, используя несколько процессорных ядер. Он начинается с запроса количества доступных ядер у операционной системы. Количество ядер, которое вы увидите, будет зависеть от количества ядер, доступных в вашем компьютере. У большинства современных компьютеров четыре или восемь ядер.

```
n = cpu_count()
print ('You have %i cores available for MapReduce' % n)
```

В данном случае вывод показывает четыре ядра (в вашем выводе может отображаться другое количество ядер):

You have 4 cores available for MapReduce



Если вы запускаете код в Windows, то по техническим причинам вы работаете с одним ядром, поэтому не сможете воспользоваться всеми доступными ядрами. Симуляция по-прежнему будет работать, но вы не увидите увеличения скорости.

Для начала код определяет операцию отображения (`map`). Затем он распределяет функцию отображения по потокам, каждый из которых обрабатывает часть исходных данных (список, содержащий слова из «Войны и мира»). Операция отображения ищет слова *peace* («мир»), *war* («война») (интересно, чего больше в «Войне и мире» — войны или мира?), *Napoleon* («Наполеон») и *Russia* («Россия»):

```
Map = partial(count_words,
               keywords=['WAR', 'PEACE', 'RUSSIA',
                        'NAPOLEON'])
map_result = Distribute(Map,
                        Partition(
                            words, len(words)//n+1), n)
print ('map_result is a list made of %i elements' %
      len(map_result))
print ('Preview of one element: %s'% map_result[0][:5])
```

Через некоторое время код выводит результаты (количество показанных элементов может варьироваться):

```
Map is a list made of 4 elements
Preview of one element: [('WAR', 1), ('PEACE', 1), ('WAR', 1),
                        ('WAR', 1), ('RUSSIA', 1)]
```

В данном случае результирующий список содержит четыре элемента, поскольку в системе четыре ядра (вы можете увидеть больше или меньше элементов в зависимости от количества ядер на вашем компьютере). Каждый элемент в списке представляет собой другой список, содержащий результаты отображения для этой части текстовых данных. Просмотрев один из этих списков, вы можете увидеть, что это последовательность связанных ключей (в зависимости от найденного ключевого слова) и единичных значений. Ключи не упорядочены, они появляются в том порядке, в котором их сгенерировал код. Следовательно, прежде чем передать списки в фазу свертки для суммирования общих результатов, код упорядочивает ключи и отправляет их на соответствующее ядро для редукции:

```
Shuffled = Shuffle_Sort(map_result)
print ('Shuffled is a list made of %i elements' %
      len(Shuffled))
print ('Preview of first element: %s'% Shuffled[0][:5])
print ('Preview of second element: %s'% Shuffled[1][:5])
```

Как показано в примере, функция **Shuffle_Sort** создает список из четырех списков, каждый из которых содержит кортежи с одним из четырех ключевых слов (ваш список может отличаться от показанного):

```
Shuffled is a list made of 4 elements
Preview of first element: [('RUSSIA', 1), ('RUSSIA', 1),
                          ('RUSSIA', 1), ('RUSSIA', 1)]
Preview of second element: [('NAPOLEON', 1), ('NAPOLEON',
                                              1), ('NAPOLEON', 1), ('NAPOLEON', 1)]
```

В кластерной среде такая обработка эквивалентна тому, что каждый узел отображения просматривает полученные результаты и, используя определенную адресацию (например, хеш-функцию, как в битовом векторе фильтра Блума из главы 12), отправляет (фаза перемешивания) данные кортежей соответствующему узлу свертки. Принимающий узел помещает каждый ключ в соответствующий список (фаза упорядочивания):

```
result = Distribute(Reduce, Shuffled, n)
print ('Emitted results are: %s' % result)
```

Фаза свертки суммирует распределенные и упорядоченные кортежи и выдает общую сумму для каждого ключа, как видно из результата, выведенного кодом, который воспроизводит MapReduce (ваш вывод может отличаться от показанного):

```
Emitted results are: [('RUSSIA', 162), ('NAPOLEON', 475),
                     ('WAR', 295), ('PEACE', 111)]
```

Анализируя результаты, вы можете увидеть, что Толстой упоминает слово «война» чаще, чем «мир», в «Войне и мире», но еще чаще он упоминает Наполеона.

Вы можете легко повторить эксперимент на других текстах или даже модифицировать функцию отображения, чтобы применить другую функцию к тексту. Например, можно проанализировать некоторые из самых известных романов сэра Артура Конан Дойля и попытаться выяснить, сколько раз Шерлок Холмс использовал фразу *Elementary, Watson* («Элементарно, Ватсон»):

```

from urllib import request

url = 'https://github.com/lmassaron/datasets/releases/'
url += 'download/1.0/1661-0.txt'
response = request.urlopen(url)
text = response.read().decode('utf-8')[932:]
words = text.split()

print (text[:60])
print ('\nTotal words are %i' % len(words))

Map = partial(count_words,
               keywords=['WATSON', 'ELEMENTARY'])
result = Distribute(Reduce,
                    Shuffle_Sort(Distribute(Map,
                                             Partition(words, len(words)//n, n)),
                             1)
print ('Emitted results are: %s' % result)

```

Результат может удивить!

```

The Adventures of Sherlock Holmes

by Arthur Conan Doyle

Total words are 107411
Emitted results are: [('WATSON', 81), ('ELEMENTARY', 1)]

```

На самом деле, эта фраза никогда не встречается в романах: это крылатое выражение сценаристы добавили позже в киносценарии (<https://www.phrases.org.uk/meanings/elementary-my-dear-watson.html>).

В ЭТОЙ ГЛАВЕ

- » Создание эффективных и умных способов кодирования
- » Использование статистики и построение деревьев Хаффмана
- » Сжатие и распаковка с использованием алгоритма Лемпеля – Зива – Велча (LZW)
- » Выполнение задач по шифрованию

ГЛАВА 14

Сжатие и сокрытие данных

За последнее десятилетие мир оказался наводнен данными. Фактически данные стали новой нефтью, и специалисты всех направлений надеются извлечь из них новые знания и богатства. В результате данные накапливаются повсюду и часто архивируются сразу после получения. Первый параграф главы посвящен использованию различных методов сжатия для эффективного хранения данных. Он разделен на две основные темы: методы кодирования и сжатие данных. Обсуждение начинается с техник кодирования.

В первом параграфе также рассматривается сжатие данных (вторая тема). Алгоритмы сжатия данных предлагают решение по уплотнению данных, позволяющее хранить больше информации на одном устройстве за счет времени обработки компьютером. Обмен дискового пространства на процессорное время снижает затраты. Сжатие также полезно в ситуациях, когда рост инфраструктуры данных не соответствует росту самих данных, что особенно актуально для беспроводной и мобильной связи в развивающихся странах. Кроме того, сжатие помогает быстрее доставлять сложные веб-страницы, эффективно передавать видео в потоковом режиме, хранить данные на мобильном устройстве или снижать затраты на передачу данных по мобильной связи. В обсуждении сжатия данных рассматривается два популярных метода сжатия: кодирование Хаффмана и алгоритм Лемпеля — Зива — Велча (LZW).

Второй параграф главы посвящен использованию кодирования для другой цели — шифрования. У сокрытия данных от тех, кто не должен их видеть, долгая история. Даже древние египтяне использовали криптографию, как объясняется на <http://www.cs.trincoll.edu/~crypto/historical/intro.html>. Конечно, современные шифры намного сложнее тех древних методов.



ЗАПОМНИТЕ

Вам не нужно вводить исходный код для этой главы вручную. На самом деле, использовать загружаемый исходный код намного проще. Исходный код главы можно найти в файлах **A4D2E; 14; Compression.ipynb** и **A4D2E; 14; Cryptography.ipynb** из загружаемых материалов. Подробнее о том, как найти эти файлы, смотрите во введении.

§ 1. Уменьшение объема данных

Компьютерные данные состоят из битов — последовательностей нулей и единиц. В главе использование нулей и единиц для создания данных объясняется подробнее, чем в предыдущих главах, поскольку сжатие использует эти нули и единицы различными способами. Чтобы понять сжатие, нужно знать, как компьютер создает и хранит бинарные числа. В следующих блоках обсуждается использование бинарных чисел в компьютерах.

Понимание кодирования

Ноль и единица — единственные числа в бинарной системе. Они представляют два возможных состояния электрической цепи: отсутствие и наличие электричества. Компьютеры начинались как простые схемы из ламп или транзисторов; использование бинарной системы вместо привычной людям десятичной упрощало задачу. Люди используют десять пальцев для счета чисел от 0 до 9. Когда нужно подсчитать больше, они добавляют разряд слева. Возможно, вы никогда не задумывались об этом, но можно выразить счет, используя степени десятки. Так, число 199 можно записать как $10^2 \times 1 + 10^1 \times 9 + 10^0 \times 9 = 199$; то есть можно разделить сотни, десятки и единицы, умножая каждую цифру на степень десятки в соответствии с ее позицией: 100 — для единиц, 101 — для десятков, 102 — для сотен и т.д.



Понимание этого помогает лучше разобраться в бинарных числах, поскольку они работают точно так же. Однако бинарные числа используют степени двойки вместо степеней десятки. Например, число 11 000 111 — это просто

$$\begin{aligned} 2^7 \cdot 1 + 2^6 \cdot 1 + 2^5 \cdot 0 + 2^4 \cdot 0 + 2^3 \cdot 0 + 2^2 \cdot 1 + 2^1 \cdot 1 + 2^0 \cdot 1 &= \\ 128 \cdot 1 + 64 \cdot 1 + 32 \cdot 0 + 16 \cdot 0 + 8 \cdot 0 + 4 \cdot 1 + 2 \cdot 1 + 1 \cdot 1 &= \\ 128 + 64 + 4 + 2 + 1 &= 199 \end{aligned}$$

Любое число можно представить в компьютере в виде бинарного значения. Значение занимает объем памяти, необходимый для его полной длины. Например, бинарное число 199 состоит из восьми цифр, каждая цифра — это бит, а 8 бит называются байтом. Компьютерное оборудование знает данные только как биты, поскольку схемы могут хранить только биты. Однако на более высоком уровне компьютерное программное обеспечение может интерпретировать биты как буквы, идеограммы, изображения, фильмы и звуки — здесь в игру вступает кодирование.

Кодирование использует последовательность битов для представления чего-то отличного от числа, выраженного самой последовательностью. Например, букву можно представить определенной последовательностью битов. Компьютерное программное обеспечение обычно представляет букву А числом 65 или бинарным 01000001 при работе со стандартом кодирования ASCII (американский стандартный код для обмена информацией). Последовательности, используемые системой ASCII, можно увидеть на сайте <https://www.asciitable.com>. ASCII использует всего 7 бит для кодирования (8 бит, или 1 байт, в расширенной версии), что означает возможность представления 128 различных символов (в расширенной версии — 256 символов). Python может представлять строку *Hello World* в байтах:

```
print (''.join(['{:0:08b}'.format(ord(l))  
               for l in "Hello World"]))
```

Этот пример выдает следующий результат:

```
01001000011001010110110001101100011011110010000001010111  
01101111011100100110110001100100
```

При использовании расширенной версии ASCII компьютер знает, что последовательность ровно из 8 бит представляет символ. Он может разделить каждую последовательность на 8-битные байты и, используя таблицу преобразования, называемую *символьной таблицей*, превратить эти байты в символы.



ЗАПОМНИТЕ

Кодировка ASCII может представлять стандартный западный алфавит, но она не поддерживает разнообразие европейских букв с диакритическими знаками или богатство неевропейских алфавитов, таких как идеограммы, используемые в китайском и японском языках. Вероятно, вы используете надежную систему кодирования, такую как UTF-8, или другую форму кодировки Unicode (подробнее см. на <https://home.unicode.org>). Кодировка Unicode — кодировка по умолчанию в Python 3.

Использование сложной системы кодирования требует применения последовательностей более длинных, чем те, что необходимы для ASCII. В зависимости от выбранной кодировки, для определения одного символа может потребоваться до 4 байт (32 бита). При представлении текстовой информации компьютер создает длинные битовые последовательности. Он легко декодирует каждую букву, поскольку кодирование использует последовательности фиксированной длины в одном файле. Стратегии кодирования, такие как UTF-8 (Unicode Transformation Format 8), могут использовать переменное количество байтов (в данном случае — от 1 до 4). Подробнее о работе UTF-8 можно прочитать на сайте <https://www.fileformat.info/info/unicode/utf8.htm>.

Рассматриваем эффекты сжатия

Использование последовательностей символов фиксированного размера оставляет большой простор для улучшений. Возможно, вы используете не все буквы алфавита или некоторые буквы встречаются чаще других. Именно здесь в игру вступает сжатие. Используя последовательности символов переменной длины, можно значительно уменьшить размер файла. Однако такой файл требует дополнительной обработки, чтобы преобразовать его обратно в несжатый формат, понятный приложениям. Сжатие удаляет пространство организованным и методичным способом; декомпрессия возвращает пространство обратно в символьные строки. Это похоже на *растворимые напитки*, которые вы покупаете в магазине: сначала это был напиток, потом порошок, а затем снова напиток. Когда возможно сжать и распаковать данные таким образом, что это не приводит к потере информации, мы имеем дело со сжатием *без потерь*.

Тот же принцип сжатия применяется к изображениям и звукам, где используются последовательности битов определенного размера для представления деталей видео или воспроизведения секунды звука через компьютерные динамики. Видео — это просто последовательности битов, и каждая битовая последовательность интерпретируется программным обеспечением как пиксель, который состоит из маленьких точек, образующих изображение. Аналогично аудио состоит из последовательностей битов, представляющих отдельные сэмплы. Аудиофайлы хранят

определенное количество сэмплов в секунду для воссоздания звука. Подробнее о видео- и аудиокодеках можно узнать на <https://www.wowza.com/blog/video-codecs-encoding> и https://developer.mozilla.org/docs/Web/Media/Formats/Audio_codecs. Компьютеры хранят данные во многих предопределенных форматах длинных последовательностей битов (обычно называемых *битовыми потоками*). Алгоритмы сжатия могут использовать особенности работы каждого формата, чтобы получить тот же результат, используя более короткий, специальный формат.

Данные, представляющие изображения и звуки, можно сжать еще сильнее, удалив детали, которые вы не можете обработать. У людей есть как визуальные, так и слуховые ограничения, поэтому они вряд ли заметят потерю деталей, возникающую при сжатии данных определенными способами. Вы наверняка слышали о сжатии MP3, которое позволяет хранить целые коллекции компакт-дисков на компьютере или портативном плеере. Формат MP3 упрощает исходный громоздкий формат WAV, используемый компьютерами. WAV-файлы содержат все звуковые волны, полученные компьютером, а MP3 экономит место, удаляя и уплотняя волны, которые вы не можете услышать. (Подробнее о MP3 можно прочитать в статье на <https://arstechnica.com/features/2007/10/the-audiofile-understanding-mp3-compression>.)



СОВЕТ

Удаление деталей из данных создает сжатие с потерями. JPEG, DjVu, MPEG, MP3 и WMA — это алгоритмы сжатия с потерями, специализированные для определенных типов медиаданных (изображения, видео, звук), и есть много других подобных алгоритмов. Сжатие с потерями приемлемо для данных, предназначенных для человеческого восприятия, однако из-за удаления деталей невозможно вернуться к исходной структуре данных. Таким образом, вы можете получить хорошее сжатие цифровой фотографии и отобразить ее приемлемым образом на экране компьютера. Однако когда вы распечатаете сжатую фотографию на бумаге, то можете заметить, что качество, хотя и приемлемое, не такое хорошее, как у оригинального снимка. Дисплей обеспечивает вывод с разрешением 96 точек на дюйм (dpi), а принтер обычно выводит изображение с разрешением от 300 до 1200 dpi (или выше). Эффекты сжатия с потерями становятся очевидными, поскольку принтер способен отображать их таким образом, что человек может их заметить.



ЗАПОМНИТЕ

Выбор между сжатием без потерь и сжатием с потерями имеет важное значение. Отбрасывание деталей — хорошая стратегия для медиафайлов, но она плохо работает с текстом, поскольку потеря слов или букв может изменить смысл текста. (По той же причине отбрасывание деталей не работает для языков программирования или компьютерных инструкций.) Хотя сжатие с потерями — это эффективное решение в случаях, когда детали не так важны, оно не подходит для ситуаций, где нужно сохранить точный смысл.

Выбор конкретного типа сжатия

Алгоритмы без потерь просто сжимают данные для уменьшения их размера и распаковывают их до исходного состояния. У алгоритмов без потерь более широкое применение, чем у сжатия с потерями, поскольку их можно использовать для любой задачи обработки данных. (Даже при использовании сжатия с потерями вы удаляете некоторые детали и дополнительно сжимаете оставшееся с помощью сжатия без потерь.) Как и в случае со множеством алгоритмов сжатия с потерями, специализированных для работы с различными медиа, есть много алгоритмов без потерь, каждый из которых умело использует определенные характеристики данных. (Чтобы получить представление о том, насколько велико семейство алгоритмов сжатия без потерь, прочитайте подробности на http://ethw.org/History_of_Lossless_Data_Compression_Algorithms.)

ПРИМЕР ПРЕИМУЩЕСТВ СЖАТИЯ С ПОТЕРЯМИ

Пример разницы, которую может дать сжатие с потерями, – это фотография. Файл фотографии в формате RAW содержит всю информацию, исходно записанную сенсором камеры, поэтому сжатие в нем отсутствует. Когда вы работаете с определенной камерой, может оказаться, что такой файл занимает 29,8 МБ на жестком диске. Файлы в формате RAW часто имеют расширение `.raw`, что указывает на отсутствие какой-либо обработки. Если открыть этот файл и сохранить его в формате с потерями, например `.jpeg`, размер файла может составить всего 3,7 МБ, но при этом произойдет соответствующая потеря детализации. Чтобы сохранить больше деталей, но при этом частично уменьшить размер файла, можно выбрать формат `.jpeg` с меньшим уровнем сжатия. В таком случае размер файла может оказаться 12,4 МБ – это будет хорошим компромиссом между экономией места на диске и сохранением качества изображения.



ЗАПОМНИТЕ

Важно помнить, что цель как сжатия с потерями, так и сжатия без потерь — уменьшить избыточность, содержащуюся в данных. Чем больше избыточности содержат данные, тем эффективнее сжатие.

Вероятно, на вашем компьютере установлено множество программ сжатия данных без потерь, создающих файлы форматов ZIP, LHA, 7-Zip и RAR, и вы не уверены, какая из них лучше. «Лучшего» варианта может не существовать, поскольку битовые последовательности можно использовать множеством различных способов для представления информации

на компьютере; кроме того, разные стратегии сжатия лучше работают с разными битовыми последовательностями. Это проблема «бесплатного обеда», обсуждавшаяся в главе 1. Выбор варианта зависит от содержимого данных, которые вам нужно сжать.

Чтобы понять, как сжатие меняется в зависимости от исходного материала, стоит попробовать различные текстовые образцы с одним и тем же алгоритмом. В следующем примере на Python используется алгоритм ZIP для сжатия текста «Приключений Шерлока Холмса» Артура Конан Дойля, а затем — для уменьшения размера случайно сгенерированной последовательности букв. (Полный код этого примера можно найти в блоке «Производительность сжатия» файла A4D2E; 14; *Compression. ipynb* в загружаемом исходном коде для этой книги; подробнее см. во введении.)

```
from urllib import request
import zlib
from random import randint

url = 'https://github.com/lmassaron/datasets/releases/'
url += 'download/1.0/1661-0.txt'
response = request.urlopen(url)
sh = response.read().decode('utf-8')[932:]
sh_length = len(sh)
rnd = ''.join([chr(randint(0, 126)) for k in
               range(sh_length)])

def zipped(text):
    return len(zlib.compress(text.encode("utf-8")))

print ("Original size for both texts: %s characters" %
      sh_length)
print ("The Adventures of Sherlock Holmes to %s" %
      zipped(sh))
print ("Random file to %s " % zipped(rnd))
```

Результаты примера весьма показательны. Хотя приложение может уменьшить размер рассказа более чем вдвое от исходного размера, для случайного текста степень сжатия оказывается значительно меньше (при том, что исходная длина обоих текстов одинакова).

```
Original size for both texts: 592905 characters
The Adventures of Sherlock Holmes to 227478
Random file to 519679
```

Это говорит о том, что алгоритм ZIP использует особенности письменного текста, но не так эффективен для случайного текста, не имеющего предсказуемой структуры.



СОВЕТ

При выполнении сжатия данных производительность можно измерить, вычислив коэффициент сжатия: достаточно разделить новый сжатый размер файла на его исходный размер. Коэффициент сжатия может рассказать об эффективности алгоритма в плане экономии пространства, однако высокопроизводительным алгоритмам тоже требуется время на выполнение задачи. Если время — критичный фактор, большинство алгоритмов позволяют пожертвовать некоторой степенью сжатия ради более быстрого сжатия и распаковки. В предыдущем примере с текстом «Шерлока Холмса» коэффициент сжатия составляет $226\,824 / 594\,941$, то есть примерно 0,381. У метода `compress()`, использованного в примере, есть второй необязательный параметр — `level`, который управляет уровнем сжатия. Изменение этого параметра позволяет контролировать соотношение между временем выполнения задачи и степенью достигаемого сжатия.

Выбор правильного кодирования

Пример из предыдущего блока показывает, что происходит при применении алгоритма ZIP к случайному тексту. Эти результаты помогают понять, почему работает сжатие. Если свести воедино все доступные алгоритмы сжатия, можно выделить четыре основные причины:

- » **Уменьшение кодировки символов.** Сжатие заставляет символы использовать меньше битов путем их кодирования в соответствии с определенными характеристиками, например частотой использования. К примеру, если вы используете только часть символов из набора символов, то можете уменьшить количество битов, чтобы отразить этот уровень использования. Это та же разница, что и между ASCII, использующей 7 бит, и расширенной ASCII, использующей 8 бит. Такое решение особенно подходит для таких задач, как кодирование ДНК, где можно разработать более эффективную схему кодирования, чем стандартная.
- » **Сжатие длинных последовательностей одинаковых битов.** При сжатии используется специальный код для идентификации множественных копий одинаковых битов, которые заменяются одной копией вместе с числом повторений. Этот метод очень эффективен для изображений

(хорошо работает с черно-белыми факсимильными изображениями) или с любыми данными, которые можно перегруппировать для объединения похожих символов (данные ДНК относятся к этой категории).

» **Использование статистики.** При сжатии часто встречающиеся символы кодируются более короткими числовыми последовательностями, чем редко встречающиеся символы, для которых используются более длинные последовательности. Например, буква *E* часто встречается в английском языке, поэтому, если для ее кодирования использовать только 3 бита вместо полных 8 бит, можно существенно сэкономить пространство. Эту стратегию использует кодирование Хаффмана, при котором пересоздается таблица символов и в среднем экономится пространство, поскольку у часто встречающихся символов более короткие коды.

» **Эффективное кодирование часто встречающихся длинных последовательностей символов.** Этот метод похож на сжатие длинных последовательностей одинаковых битов, но работает с последовательностями символов, а не с отдельными символами. Такую стратегию использует алгоритм LZW, который изучает шаблоны данных «на лету» создает короткие коды для длинных последовательностей символов.

Чтобы понять, как переосмысление кодирования может помочь в сжатии, начнем с первой причины. Ученые, работавшие над проектом «Геном» примерно в 2008 году (<https://www.genome.gov/human-genome-project>), смогли радикально уменьшить размер данных, используя простой трюк кодирования. Применение этого приема упростило задачу картирования всей человеческой ДНК, помогая ученым лучше понять жизнь, болезни и смерть, записанные в клетках нашего тела.

Ученые описывают ДНК с помощью последовательностей букв *A*, *C*, *T* и *G* (представляющих четыре нуклеотида, присутствующих во всех живых существах). Человеческий геном содержит 6 миллиардов нуклеотидов (они встречаются парами, называемыми основаниями), что при кодировании ASCII составляет более 50 ГБ. Фактически *A*, *C*, *T* и *G* в кодировке ASCII представляются следующим образом:

```
print (' '.join(['{0:08b}'.format(ord(l))
                 for l in «ACTG»]))

01000001 01000011 01010100 01000111
```


Сумма предыдущей строки составляет 32 бита, но, поскольку ДНК использует всего четыре символа, можно использовать по 2 бита на каждый, экономя 75% ранее используемых битов:

```
00 01 10 11
```



ЗАПОМНИТЕ

Такой выигрыш демонстрирует важность выбора правильного кодирования. В данном случае кодирование работает хорошо, потому что алфавит ДНК состоит из четырех букв, и использование полной таблицы ASCII на основе 8 бит избыточно. Если задача требует использования полного алфавита ASCII, сжать данные путем переопределения используемого кодирования не получится. Вместо этого вам нужно подойти к проблеме, используя сжатие Хаффмана.

Если вы не можете уменьшить кодировку символов (или уже сделали это), вы все еще можете сжать длинные последовательности, сведя их к более простому кодированию. Посмотрите, как в бинарных данных могут повторяться длинные последовательности единиц и нулей:

```
00000000 00000000 01111111 11111111 10000011 11111111
```

В этом случае последовательность начинается с нуля. Поэтому вы можете подсчитать количество нулей, затем количество следующих за ними единиц и так далее, для следующего количества нулей. Поскольку последовательность содержит только нули и единицы, вы можете подсчитать их и получить последовательность чисел для сжатия данных. В этом случае данные сжимаются в значения 17 15 5 10. Преобразование этих чисел в байты сокращает исходные данные легкообратимым способом:

```
00010001 00001111 00000101 00001010
```

Вместо использования 6 байт для представления данных теперь требуется только 4 байта. Чтобы использовать этот подход, максимальное количество последовательных значений ограничивается 255, что означает:

- » вы можете закодировать каждую последовательность в один байт;
- » первое значение – ноль, когда последовательность начинается с 1, а не с 0;
- » когда блок значений длиннее 255 элементов, вы вставляете значение 0 (чтобы декодер переключился на другое значение для нулевых счетчиков и затем начал снова подсчитывать первое значение).

Этот алгоритм, *кодирование длин повторов* (RLE), очень эффективен, если ваши данные содержат много длинных повторов. Этот алгоритм имел большой успех в 1980-х годах, поскольку мог сократить время передачи факсов. Факсимильные аппараты работали только с черно-белыми изображениями по телефонным линиям, поэтому сжатие длинных последовательностей нулей и единиц, составляющих изображения и текст, оказалось удобным. Хотя сейчас предприятия редко используют факсы, ученые все еще применяют RLE для сжатия ДНК в сочетании с преобразованием Барроуза — Уилера (продвинутый алгоритм, о котором можно прочитать на <https://marknelson.us/posts/1996/09/01/bwt.html>), которое переупорядочивает (обратимым способом) геномную последовательность в длинные цепочки одинаковых нуклеотидов. RLE также используется для сжатия других форматов данных, таких как JPEG и MPEG (дополнительные подробности см. на <https://motorscript.com/mpeg-jpeg-compression>).



СОВЕТ

Характеристики данных определяют успех алгоритма сжатия. Зная, как работают алгоритмы, и исследуя характеристики ваших данных, вы можете выбрать наиболее эффективный алгоритм или эффективно комбинировать несколько алгоритмов. Совместное использование нескольких алгоритмов создает *ансамбль алгоритмов*.

Кодирование с использованием сжатия Хаффмана

Переопределение кодировки, например при отображении нуклеотидов в ДНК, — это умное решение, которое работает, только когда вы используете часть алфавита, представленного кодировкой. Когда вы используете все символы в кодировке, это решение неприменимо. Дэвид А. Хаффман открыл другой способ эффективного кодирования букв, цифр и символов, даже при использовании их всех. Он достиг этого результата, будучи студентом MIT в 1952 году, в рамках курсовой работы, заданной его профессором Робертом М. Фано. Его профессор и другой известный ученый, Клод Шеннон (отец теории информации), тоже бились над этой проблемой.

В статье «Метод построения кодов с минимальной избыточностью» Хаффман всего на трех страницах описывает свой потрясающий метод кодирования. Он изменил способ хранения данных вплоть до конца 1990-х годов. Подробнее об этом удивительном алгоритме можно прочитать в статье журнала *Scientific American* за сентябрь 1991 года по адресу <http://www.huffmancoding.com/my-uncle/scientific-american>.

В основе кодов Хаффмана лежит три ключевые идеи:

- » **кодирование частых символов более короткими последовательностями битов.** Например, если в вашем тексте часто встречается буква *a*, но редко используется буква *z*, вы можете закодировать *a* парой битов, а для *z* — зарезервировать целый байт (или больше). Использование более коротких последовательностей для часто встречающихся букв означает, что в целом ваш текст будет занимать меньше байтов, чем при использовании ASCII-кодировки;
- » **кодирование коротких последовательностей уникальными сериями битов.** При использовании битовых последовательностей переменной длины нужно гарантировать, что короткая последовательность не будет ошибочно принята за более длинную из-за их схожести. Например, если буква *a* в бинарном виде — 110, а *z* — 110110, то букву *z* можно ошибочно интерпретировать как последовательность из двух букв *a*. Кодирование Хаффмана избегает этой проблемы, используя префиксные коды: алгоритм никогда не использует более короткие последовательности в качестве начальных частей более длинных. Например, если *a* — это 110, то *z* будет 101110, а не 110110;
- » **управление префиксным кодированием с помощью специальной стратегии.** Кодирование Хаффмана управляет префиксными кодами, используя бинарные деревья особым образом. Бинарные деревья — это структура данных, которая обсуждается в главах 6 и 7. Алгоритм Хаффмана использует бинарные деревья (называемые деревьями Хаффмана) на продвинутом уровне. Подробнее о внутреннем устройстве алгоритма можно прочитать в руководстве по адресу <https://www.tutorialspoint.com/huffman-trees-in-data-structure>.

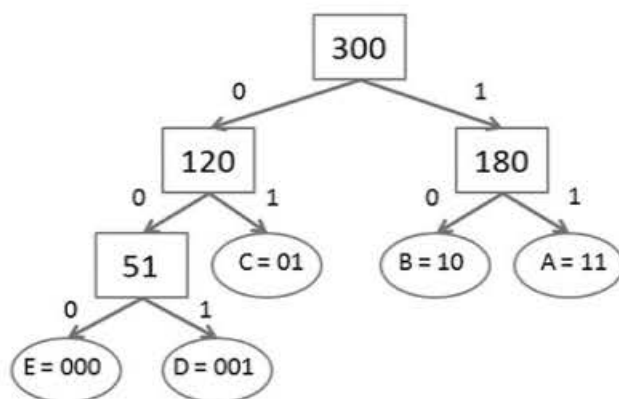


ПРИМЕЧАНИЕ

Алгоритм, используемый для кодирования Хаффмана, представляет собой итеративный процесс, основанный на *кучах* — специализированных древовидных структурах данных (упомянутых в главе 6). Куча — это сложная структура данных. Благодаря способу организации данных в куче, она полезна для реализации *жадной стратегии*. В следующей главе, посвященной жадным алгоритмам, вы сами протестируете кодирование Хаффмана, используя рабочие примеры из загружаемого кода, сопровождающего книгу (пример *Huffman Compression* — в файле **A4D2E; 15; Greedy Algorithms.ipynb**; подробнее о том, где найти этот файл, см. во введении).

В качестве примера результата кодирования Хаффмана на рисунке 14.1 показано бинарное дерево Хаффмана, используемое для кодирования длинной последовательности букв *ABCDE*, распределенных таким образом, что *A* встречается чаще, чем *B*, *B* — чаще, чем *C*, *C* — чаще, чем *D*, а *D* — чаще, чем *E*.

РИСУНОК 14.1.
Дерево Хаффмана и его таблица символического преобразования



Квадратные узлы представляют собой узлы *ветвления*, где алгоритм размещает количество оставшихся букв, которые он распределяет между *дочерними узлами* (расположенными ниже узлов ветвления в иерархии). Круглые узлы представляют собой *листовые узлы*, где находятся успешно закодированные буквы. Дерево начинается с *корня*, содержащего 300 букв для распределения (длина текста). Оно распределяет буквы, ветвясь битами 0 и 1 соответственно по левым и правым ветвям, пока не достигнет всех необходимых для кодирования листьев. Читая последовательность ветвей от вершины до конкретной буквы, вы определяете бинарную последовательность, представляющую эту букву. Менее частые буквы (*D* и *E*) получают самые длинные бинарные последовательности.



СОВЕТ

Следуя по дереву Хаффмана снизу вверх, можно сжать символ в бинарную последовательность. Двигаясь по дереву сверху вниз, можно распаковать бинарную последовательность в символ (представленный первым встреченным листовым узлом).



ЗАПОМНИТЕ

Для распаковки нужно хранить как сжатую бинарную последовательность, так и дерево Хаффмана, сделавшее сжатие возможным. Когда текст или данные слишком короткие, дерево Хаффмана может занимать больше места, чем сжатые данные, делая сжатие неэффективным. Код Хаффмана лучше всего работает с более крупными файлами данных.

Запоминание последовательностей с помощью LZW

Кодирование Хаффмана использует преимущество наиболее часто встречающихся символов, чисел или знаков в данных и сокращает их кодировку. Алгоритм LZW выполняет аналогичную задачу, но расширяет процесс кодирования на наиболее частые последовательности символов. Алгоритм LZW появился в 1984 году и был создан Абрахамом Лемпелем, Якобом Зивом и Терри Велчем на основе более раннего алгоритма LZ78 (разработанного в 1978 году только Лемпелем и Зивом). Как Unix-сжатие, так и формат изображений GIF основаны на этом алгоритме. LZW использует повторения, поэтому он идеально подходит и для сжатия документов и книжных текстов, поскольку люди часто используют одни и те же слова при написании. Кроме того, LZW может работать с потоковыми данными, а алгоритм Хаффмана — нет; алгоритму Хаффмана нужен полный набор данных для построения таблицы отображения.

Просматривая поток битов данных, алгоритм изучает последовательности символов и назначает каждой последовательности короткий код. Таким образом, при повторной встрече той же серии символов LZW может сжать их, используя более простую кодировку. Интересно, что этот алгоритм начинает работу с таблицы символов, состоящей из одиночных символов (обычно таблицы ASCII), а затем расширяет эту таблицу, используя последовательности символов, которые он изучает из сжимаемых данных.

Более того, LZW не требует хранить изученные последовательности в таблице для распаковки — он может легко восстановить их, считывая сжатые данные. LZW способен реконструировать шаги, которые он выполнял при сжатии исходных данных, и закодированные последовательности. У этой возможности своя цена: поначалу LZW неэффективен. Он лучше всего работает с большими фрагментами данных или текста (что характерно и для других алгоритмов сжатия).

LZW — несложный алгоритм, но, чтобы полностью его понять, нужно рассмотреть несколько примеров. Вы можете найти довольно много хороших руководств по адресам <https://marknelson.us/posts/2011/11/08/lzw-revisited.html> и http://www.matthewflickinger.com/lab/whatsinagif/lzw_image_data.asp. Второе руководство объясняет, как использовать LZW для сжатия изображений. В следующем примере показана реализация на Python. (Полный код этого примера вы найдете в разделе *LZW* файла `A4D2E: 14; Compression.ipynb` из загружаемого исходного кода для этой книги; подробнее см. во введении).

```
def lzw_compress(text):
    dictionary = {chr(k): k for k in range(256)}
    encoded = list()
```

```

s = text[0]
for c in text[1:]:
    if s+c in dictionary:
        s = s+c
    else:
        print ('> %s' %s)
        encoded.append(dictionary[s])
        print ('found: %s compressed as %s' %
              (s,dictionary[s]))
        dictionary[s+c] = max(dictionary.values()) + 1
        print ('New sequence %s indexed as %s' %
              (s+c, dictionary[s+c]))
        s = s+c
encoded.append(dictionary[s])
print ('found: %s compressed as %s'
      %(s,dictionary[s]))
return encoded

```

В этом примере алгоритм сканирует текст, проверяя его посимвольно. Он начинает с кодирования символов, используя исходную таблицу символов, которая в данном случае фактически является таблицей ASCII. Лучший способ понять, как работает этот код, — посмотреть серию выходных сообщений и проанализировать, что произошло, как показано здесь:

```

text = "ABABCABCABC"
compressed = lzw_compress(text)
print('\nCompressed: %s \n' % compressed)

```

Как показывает вывод, данные действительно сжаты:

```

> A
found: A compressed as 65
New sequence AB indexed as 256
> B
found: B compressed as 66
New sequence BA indexed as 257
> AB
found: AB compressed as 256
New sequence ABC indexed as 258
> C
found: C compressed as 67
New sequence CA indexed as 259
> ABC
found: ABC compressed as 258
New sequence ABCA indexed as 260
found: ABC compressed as 258

```


Вот краткое описание того, что означают эти выходные сообщения:

- 1 Первая буква — *A* — присутствует в исходной таблице символов, поэтому алгоритм кодирует ее как 65.
- 2 Вторая буква — *B* — отличается от *A*, но тоже присутствует в исходной таблице символов, поэтому алгоритм кодирует ее как 66.
- 3 Третья буква — снова *A*, поэтому алгоритм читает следующую букву, которой является *B*, и кодирует эту двухбуквенную комбинацию *AB* как 256.
- 4 Четвертая буква — *C* — отличается от всех предыдущих букв и тоже присутствует в исходной таблице символов, поэтому алгоритм кодирует ее как 67.
- 5 Следующая буква уже встречалась ранее — это *A*. Следующая буква — *B*, что дает комбинацию букв *AB*; она тоже присутствует в символьной таблице. Однако следующая буква — *C*, что создает новую последовательность, которую алгоритм теперь кодирует как 258.
- 6 Последние три буквы представляют собой еще один набор *ABC*, поэтому их код снова равен 258. Следовательно, закодированный вывод для *ABABCSABCSABC* состоит из:

Compressed: [65, 66, 256, 67, 258, 258]

Все операции обучения и кодирования преобразуются в окончательные сжатые данные, состоящие всего из шести числовых кодов (каждый — по 8 бит) против исходных 11 тестовых букв. Кодирование дает хороший коэффициент сжатия, примерно половину от исходных данных: $6/11 = 0,55$.

Восстановление исходного текста из сжатых данных требует другой, обратной процедуры, которая учитывает единственную ситуацию, когда декодирование LZW может не суметь реконструировать символьную таблицу — когда последовательность начинается и заканчивается одним и тем же символом. Этот особый случай обрабатывается в Python с помощью блока команд `if — then — else`, поэтому вы можете безопасно использовать алгоритм для кодирования и декодирования чего угодно:

```
def lzw_decompress(encoded):
    reverse_dictionary = {k:chr(k) for k in range(256)}
    current = encoded[0]
    output = reverse_dictionary[current]
    print ('Decompressed %s ' % output)
    print ('>%s' % output)
    for element in encoded[1:]:
        previous = current
```

```

current = element
if current in reverse_dictionary:
    s = reverse_dictionary[current]
    print ('Decompressed %s ' % s)
    output += s
    print ('>%s' % output)
    new_index = max(reverse_dictionary.keys()) + 1
    reverse_dictionary[new_index
] = reverse_dictionary[previous] + s[0]
    print ('New dictionary entry %s at index %s' %
          (reverse_dictionary[previous] + s[0],
           new_index))
else:
    print ('Not found:',current,'Output:',
          reverse_dictionary[previous
] + reverse_dictionary[previous][0])
    s = reverse_dictionary[previous
] + reverse_dictionary[previous][0]
    print ('New dictionary entry %s at index %s' %
          (s, max(reverse_dictionary.keys())+1))
    reverse_dictionary[
        max(reverse_dictionary.keys())+1] = s
    print ('Decompressed %s' % s)
    output += s
    print ('>%s' % output)
return output

```

Запуск функции на ранее сжатой последовательности восстанавливает исходную информацию путем сканирования символьной таблицы, как показано здесь:

```

print ('\ndecompressed string: %s' %
      lzw_decompress(compressed))
print ('original string was: %s' % text)
Decompressed A
> A
Decompressed B
> AB
New dictionary entry AB at index 256
Decompressed AB
> ABAB
New dictionary entry BA at index 257
Decompressed C
> ABABC
New dictionary entry ABC at index 258
Decompressed ABC

```

```
> ABABCABC
New dictionary entry CA at index 259
Decompressed ABC
> ABABCABCABC
New dictionary entry ABCA at index 260

decompressed string: ABABCABCABC
original string was: ABABCABCABC
```

§ 2. Соккрытие ваших секретов с помощью криптографии

В эпоху, когда каждый, кажется, вмешивается в дела других, *криптография* — искусство делать текст нечитаемым так, чтобы посвященные могли позже его восстановить, — приобрела особое значение. Криптография нужна для того, чтобы держать секретный текст подальше от людей, которым не нужно о нем знать. Как упоминалось ранее в главе, криптография — это древнее искусство (поскольку нужно быть креативным, чтобы определить уникальный криптографический метод) и наука (поскольку криптография должна быть обратимой), берущая начало у египтян. Основы криптографии не менялись тысячелетиями. Процесс:

- 1 использует *открытый текст* (исходные данные);
- 2 *шифрует* его (делает нечитаемым);
- 3 отправляет данные получателю в виде *шифротекста* (который нечитаем);
- 4 *расшифровывает* его (делает снова читаемым).

Современная криптография опирается на всевозможные продвинутые математические методы для выполнения шагов шифрования и расшифровки. Потребовалось бы несколько книг (или даже больше), чтобы обсудить эту тему хоть сколько-нибудь глубоко, и к тому времени ваш мозг превратился бы в чистое желе (это случается со всеми). В следующих блоках дается краткий обзор криптографии — достаточный, чтобы блеснуть умом на следующей офисной вечеринке, но недостаточный, чтобы превратить ваш мозг в желе.



ВНИМАНИЕ

Примеры в следующих блоках — учебные, предназначенные для обучения принципам, а не для создания нераскрываемого криптографического вывода. Следовательно, вы должны использовать эти примеры для понимания того, как работает криптография, а не шифровать свое следующее секретное послание лучшему другу в надежде, что никто другой не сможет его прочесть. В конце концов, любой, кто прочтет эту книгу, сможет прочитать ваше сообщение и рассказать об этом всему миру!

Подстановка символов

Подстановка символов — один из самых простых и быстрых методов шифрования данных. Еще это один из самых взламываемых методов, поскольку такая форма шифрования не очень хорошо скрывает естественные шаблоны данных (что подтверждается теми книгами с головоломками, которые позволяют решать такое шифрование, используя только распознавание шаблонов). Один из первых методов подстановки символов был разработан Юлием Цезарем, и вы можете прочитать о нем на <https://www.dcode.fr/caesar-cipher>. Тем не менее, проявив немного изобретательности, вы можете создать шифр подстановки символов с помощью Python, обеспечивающего переменное шифрование:

```
import random

random.seed(input("Provide an encryption value: "))

letters = ["a", "b", "c", "d", "e", "f", "g", "h", "i",
           "j", "k", "l", "m", "n", "o", "p", "q", "r",
           "s", "t", "u", "v", "w", "x", "y", "z", "!",
           "A", "B", "C", "D", "E", "F", "G", "H", "I",
           "J", "K", "L", "M", "N", "O", "P", "Q", "R",
           "S", "T", "U", "V", "W", "X", "Y", "Z", " "]

randomLetters = letters.copy()
random.shuffle(randomLetters)
print(letters)
print(randomLetters)
```

Пример начинается со списка букв, пробела и восклицательного знака. Он копирует эти буквы в `randomLetters`, а затем перемешивает их на основе предоставленного вами значения `random.seed()`. Значение начального числа — это *ключ*, секретное значение, известное только вам. Поскольку ключ не зашит в приложении жестко, вам нужно знать его для последующей расшифровки входной строки. При запуске этого кода с начальным значением 5 вы увидите следующий результат:

```

Provide an encryption value: 5
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l',
 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v',
 'w', 'x', 'y', 'z', '!', 'A', 'B', 'C', 'D', 'E',
 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y',
 'Z', ' ']
['x', 'q', 'i', 'y', 'l', 'B', 'u', 'a', 'z', 'L', 'v', 'A',
 'P', 'O', 'X', 'e', 'c', 'D', 't', 'f', 'Z', 'p',
 'k', 'V', 'm', 'S', 's', '!', ' ', 'H', 'R', 'Q',
 'U', 'E', 'h', 'j', 'Y', 'n', 'b', 'I', 'M', 'J',
 'g', 'N', 'K', 'C', 'o', 'F', 'd', 'T', 'G', 'W',
 'r', 'w']

```

Этот пример не использует сложных алгоритмов для шифрования и дешифрования входной строки, но шаги похожи на шаги любого другого алгоритма. Чтобы расшифровать строку, нужно выполнить действия обратные тем, что были сделаны при шифровании, как показано здесь:

```

def encrypt(text):
    result=""
    for i in range(len(text)):
        char = text[i]
        result += randomLetters[letters.index(char)]
    return result

def decrypt(text):
    result=""
    for i in range(len(text)):
        char = text[i]
        result += letters[randomLetters.index(char)]
    return result

```

Алгоритм использует простую подстановку для выполнения задачи. Соответственно, когда вы запускаете этот код:

```

encrypted = encrypt("Hello There!")
print(encrypted)

```

вы видите результат точно такой же длины, как и входная строка, но довольно сложный для чтения:

```

hIAAXwoalDls

```

При дешифровании процесс должен убедиться, что вы создатель зашифрованной строки, поэтому снова запрашивает начальное значение, как показано здесь:

```
random.seed(input("Provide a decryption value: "))
randomLetters = letters.copy()
random.shuffle(randomLetters)

decrypted = decrypt(encrypted)
print(decrypted)
```

Ввод неправильного начального значения, неверного ключа, приведет к некорректной расшифровке строки. Этот процесс очень похож на тот, что используется в более сложных методах шифрования. Разница заключается в алгоритме. Например, стандарт шифрования Advanced Encryption Standard (AES) использует невероятно сложный алгоритм, описанный на странице <https://www.geeksforgeeks.org/advanced-encryption-standard-aes>. Другие примеры методов шифрования на Python можно найти по адресу https://www.tutorialspoint.com/cryptography_with_python/cryptography_with_python_quick_guide.htm.

Работа с шифрованием AES

Теперь, когда вы увидели базовый пример, пришло время рассмотреть код, который можно использовать в реальных условиях. Однако, прежде чем использовать этот пример, вам нужно установить пакет **PyCrypto** (<https://pypi.org/project/pycrypto>) с помощью следующей команды в командной строке Anaconda (не используйте обычную командную строку; возможно, придется открыть командную строку Anaconda с правами администратора):

```
conda install pycrypto
```

Пример в этом блоке использует процесс, очень похожий на тот, что был в предыдущем блоке, но алгоритм — сложнее, а результаты гораздо труднее взломать. Код шифрования и дешифрования выглядит так:

```
import base64, re
from Crypto.Cipher import AES
from Crypto import Random

def encrypt(key, blk_sz, raw):
    raw = raw + '\0' * (blk_sz - len(raw) % blk_sz)
    raw = raw.encode('utf-8')
    secretKey = Random.new().read(AES.block_size)
```



```

cipher = AES.new(key.encode('utf-8'), AES.MODE_CBC,
                  secretKey)
return base64.b64encode(
    secretKey + cipher.encrypt(raw) ).decode(
    'utf-8')

def decrypt(key, enc):
    enc = base64.b64decode(enc)
    secretKey = enc[:16]
    cipher = AES.new(key.encode('utf-8'), AES.MODE_CBC,
                      secretKey)
    return re.sub(b'\x00*$', b'',
                  cipher.decrypt(enc[16:])).decode(
    'utf-8')

```

ИСПРАВЛЕНИЕ ОШИБКИ `TIME.CLOCK()`

После установки **PyCrypto**, возможно, потребуется выполнить дополнительное действие. Вызов `time.clock()` устарел в Python 3.3 и был удален в Python 3.8 и выше. Если при запуске примера вы получаете ошибку `AttributeError: module 'time' has no attribute 'clock'`, то решение довольно простое. Подробнее об этом можно прочитать на странице <https://stackoverflow.com/questions/58569361/attributeerror-module-time-has-no-attribute-clock-in-python-3-8>. В сущности, вам нужно открыть файл `C:\Users\<Your Name>\anaconda3\Lib\site-packages\PyCrypto\Random_UserFriendlyRNG.py` на вашем компьютере и найти строку `t = time.clock()`. Замените каждое такое вхождение на `t = time.time()` и сохраните файл. Проблема должна быть устранена в одном из следующих обновлений библиотеки PyCrypto.

Процесс шифрования начинается с *дополнения* (padding) текста для рас-шифровки нулями в конце, чтобы он равномерно делился на размер блока. Например, если исходное сообщение содержит 25 символов (а размер блока — 32 символа), этот процесс дополнения добавляет семь нулей, чтобы сделать сообщение длиной 32 символа. Текст также кодируется в UTF-8, поэтому нули отображаются не как управляющий символ 0, а как `\x00`. Причина, по которой нужно дополнять сообщение, заключается в том, что в примере используется блочный шифр.

Следующий шаг процесса генерирует секретный ключ для шифрования сообщения, как описано на странице <https://www.dlitz.net/software/pycrypto/doc/#crypto-publickey-public-key-algorithms>. Этот секретный

ключ противопоставляется открытому ключу, который вы предоставляете при вызове функции `encrypt()`. Поскольку секретный ключ неизвестен, его практически невозможно угадать, чтобы раскрыть содержимое сообщения без наличия открытого ключа (того, который известен). Затем секретный ключ используется для кодирования сообщения с помощью `AES.new()` (описано на странице <https://pycryptodome.readthedocs.io/en/latest/src/cipher/aes.html>) и вывода его в виде шифротекста. В завершение функция `encrypt()` выводит комбинацию секретного ключа (используемого позже для расшифровки сообщения) и шифротекста, декодированного в формате UTF-8 и закодированного в формате base64.



ПРИМЕЧАНИЕ

Кодировка base64 гарантирует, что строки остаются в читаемом для человека виде и не зависят от различий между платформами в обработке специальных типов символов, таких как управляющие символы. Подробнее о кодировке base64 можно прочитать на странице <https://levelup.gitconnected.com/what-is-base64-encoding-4b5ed1eb58a4>. Сайты <https://www.base64encode.org> и <https://www.base64decode.org> демонстрируют, как работает кодировка base64.

Расшифровка следует процессу, в некотором роде противоположному шифрованию. Код начинается с декодирования зашифрованного в base64 сообщения. Затем он извлекает секретный ключ из первых 16 символов комбинированного сообщения и использует `key` и `secretKey` для создания объекта шифрования, который затем используется для расшифровки сообщения. Вызов `re.sub()` (см. на <https://docs.python.org/3/library/re.html>) удаляет ранее добавленное дополнение, и в результате вы получаете исходное сообщение. Следующий код показывает, как использовать эти две функции:

```
encryp_msg = encrypt("1234567890ABCDEF", 16,
                    "This is a secret message.")
print(encryp_msg)

msg = decrypt("1234567890ABCDEF", encryp_msg)
print(msg)
```

Сначала код вызывает `encrypt()` с открытым ключом, размером блока и сообщением для шифрования. Обратите внимание, что длина открытого ключа — 16 символов. Вы не можете использовать ключ меньшего размера, не дополнив его до 16 символов, поскольку в данном случае размер блока составляет 16 символов. При запуске этого кода вы увидите перемешанный вывод, похожий на этот:

```
r6KPOR0pbHqE7aZzCpF4wVgV6/YrakgD7pSwHBVTPYUrOwmZpbknqZADtxOxBN
```

Вызов `decrypt()` снова использует открытый ключ вместе с зашифрованным сообщением. При выполнении этого кода вы снова увидите исходное сообщение.



**РЕШЕНИЕ
СЛОЖНЫХ
ЗАДАЧ**

В ЭТОМ РАЗДЕЛЕ

Использование жадных, динамических
и рандомизированных алгоритмов

Поиск компромиссных решений с помощью
локального поиска

Применение методов линейного
программирования

Изучение различных видов эвристик

В ЭТОЙ ГЛАВЕ

- » Изучение того, как проектировать новые алгоритмы и использовать различные парадигмы решения задач
- » Объяснение того, как жадный алгоритм может давать отличные результаты
- » Проектирование собственного жадного алгоритма
- » Возвращение к кодированию Хаффмана и рассмотрение других классических примеров

ГЛАВА 15

Работа с жадными алгоритмами

После первых шагов в мир алгоритмов, где мы рассмотрели, что такое алгоритмы, и обсудили упорядочивание, поиск, графы и большие данные, пришло время перейти к более общей части книги. В этой заключительной части книги вы столкнетесь с некоторыми сложными примерами и познакомитесь с общими алгоритмическими подходами, которые можно применять в различных обстоятельствах при решении реальных задач.

Исследуя новые пути и подходы, эта глава выходит далеко за рамки рекурсивного подхода «разделяй и властвуй», который доминирует в большинстве задач сортировки. Некоторые из обсуждаемых решений не являются полностью новыми — вы уже видели их в предыдущих главах. Однако в этой главе эти алгоритмы рассматриваются глубже, в контексте новых *парадигм*. Эти парадигмы включают рассмотрение правил и условий применения, общий подход и шаги к решению задачи, а также анализ сложности проблемы, ограничений и подводных камней.

Обобщение некоторых решений и описание их как широко применимых парадигм — это способ предложить подсказки для решения новых практических задач, что является частью анализа и проектирования алгоритмов. В оставшейся части книги рассматриваются другие общие подходы.



ЗАПОМНИТЕ

Вам не нужно вводить исходный код для этой главы вручную. На самом деле, использовать загружаемый исходный код намного проще. Исходный код для главы можно найти в файле `A4D2E; 15; Greedy Algorithms.ipynb` из загружаемых материалов. Подробнее о том, как найти этот файл, смотрите во введении.

§ 1. Решаем, когда лучше быть жадным

Жадные алгоритмы оказываются полезными при решении широкого спектра задач, особенно когда разработка глобального решения затруднена. Иногда стоит отказаться от сложных планов и просто начать искать легкодоступные решения, похожие на те, что вам нужны. В алгоритмах такой недалёковидный подход можно назвать *жадным* (в отличие от тех случаев, когда вы хотите второй десерт, что тоже можно считать жадностью). Поиск легкодостижимых решений составляет основную отличительную характеристику жадных алгоритмов. Жадный алгоритм достигает решения задачи с помощью последовательных шагов, на каждом из которых он принимает решение на основе лучшего решения в данную минуту, не учитывая последствия.



ЗАПОМНИТЕ

Два элемента являются ключевыми для определения жадного алгоритма:

- » на каждом шаге вы всегда принимаете наилучшее возможное решение в данную конкретную минуту;
- » вы надеетесь, что серия наилучших решений приведет к оптимальному конечному результату.

Жадные алгоритмы просты, интуитивно понятны, компактны и быстры, поскольку обычно работают за *линейное время* (время выполнения пропорционально количеству входных данных). К сожалению, они не всегда дают оптимальное решение для всех задач, но когда это происходит, они быстро предоставляют наилучшие результаты. Даже когда они не дают оптимального ответа, они могут предложить неоптимальное решение, которое может быть достаточным или использоваться как отправная точка для дальнейшего улучшения с помощью другой алгоритмической стратегии. Это похоже на ситуации, когда нужно быстро принять решение, которое будет «достаточно хорошим».

Интересно, что жадные алгоритмы напоминают то, как люди решают многие простые задачи, не прилагая особых умственных усилий или имея ограниченную информацию. Например, когда кассиры выдают сдачу, они естественным образом используют жадный подход. Задачу *выдачи сдачи* можно сформулировать как выплату определенной суммы (сдачи) с использованием наименьшего количества купюр и монет из доступных номиналов. Следующий пример на Python демонстрирует решение задачи выдачи сдачи с использованием жадного подхода:

```
def change(to_be_changed, denomination):
    resulting_change = list()
    for bill in denomination:
        while to_be_changed >= bill:
            resulting_change.append(bill)
            to_be_changed = to_be_changed - bill
    return resulting_change, len(resulting_change)

currency = [100, 50, 20, 10, 5, 1]
amount = 367
print ('Change: %s (using %i bills)'
      % (change(amount, currency)))

Change: [100, 100, 100, 50, 10, 5, 1, 1] (using 8 bills)
```

Алгоритм, заключенный в функции `change()`, просматривает доступные номиналы от большего к меньшему. Он использует самую крупную доступную купюру для выдачи сдачи, пока оставшаяся сумма не станет меньше этого номинала. Затем он переходит к следующему номиналу и выполняет ту же операцию, пока не дойдет до самого мелкого номинала. Таким образом, `change()` всегда выдает самую крупную возможную купюру для выплачиваемой суммы (это и есть жадный принцип в действии).

Жадные алгоритмы ценятся при решении задач планирования, оптимального кэширования (улучшение работы операционной системы компьютера и веб-браузера) и сжатия с использованием кодирования Хаффмана. Они также хорошо работают для некоторых задач на графах. Например, алгоритмы Краскала и Прима для поиска минимального остовного дерева и алгоритм Дейкстры для поиска кратчайшего пути — все они жадные (подробности — в главе 9). Жадный подход также может предложить неоптимальное, но приемлемое первое приближенное решение для сложных задач, таких как задача коммивояжера (traveling salesman problem, TSP). Кроме того, жадные алгоритмы могут решать особые случаи задач, которые в остальном решаются более сложными алгоритмами, например задачу о рюкзаке, когда количества не являются дискретными (обе задачи рассматриваются в главе 16).

Почему жадный подход эффективен

Неудивительно, что жадная стратегия так хорошо работает в задаче выдачи сдачи. Фактически некоторые задачи не требуют дальновидных стратегий: решение строится с использованием промежуточных результатов (последовательности решений) и на каждом шаге правильное решение всегда является наилучшим согласно изначально выбранному критерию.

«Жадное» поведение также является очень человеческим (и эффективным) подходом к решению экономических проблем. В фильме 1987 года *«Уолл-стрит»* главный герой Гордон Гекко заявляет, что «жадность, за неимением лучшего слова, — это хорошо», и превозносит жадность как положительное явление в экономике. Жадность (не в моральном смысле, а в смысле использования жадного алгоритма) лежит в основе неоклассической экономики. Экономисты, такие как Адам Смит, в XVIII веке теоретизировали, что стремление индивида к собственной выгоде приносит значительную пользу обществу в целом и делает его процветающим в экономическом плане (см.: <https://plus.maths.org/content/adam-smith-and-invisible-hand>).

Детальное описание работы жадного алгоритма (и условий, при которых он может работать корректно) довольно простое и включает следующие четыре шага:

- 1 Проблему можно разделить на частичные задачи. Сумма (или другая комбинация) этих частичных задач дает правильное решение. В этом смысле жадный алгоритм не сильно отличается от алгоритма «разделяй и властвуй» (как быстрая сортировка или сортировка слиянием, которые рассматриваются в главе 7).
- 2 Успешное выполнение алгоритма зависит от успешного выполнения каждого частичного шага. Это характеристика оптимальной подструктуры, поскольку оптимальное решение состоит только из *оптимальных подрешений*. Убедиться в том, что у задачи оптимальная подструктура, можно аналитически или путем систематического отслеживания от оптимального решения.
- 3 Для достижения успеха на каждом шаге алгоритм учитывает входные данные только на этом шаге. То есть статус ситуации (предыдущие решения) определяет решение, которое принимает алгоритм, но алгоритм не рассматривает последствия. Это полное отсутствие глобальной стратегии — *свойство жадного выбора*, поскольку жадности на каждом этапе достаточно для достижения конечного успеха. Как аналогия: это похоже на игру в шахматы, когда вы не смотрите дальше одного хода вперед и тем не менее выигрываете партию.

- 4 Свойство жадного выбора дает надежду на успех, поэтому жадный алгоритм не требует сложного правила принятия решений, поскольку ему нужно в худшем случае рассмотреть все доступные входные элементы на каждом этапе. Нет необходимости вычислять возможные последствия решений; следовательно, вычислительная сложность — в худшем случае линейная $O(n)$. Жадные алгоритмы выбирают простой путь решения очень сложных задач, которые другие алгоритмы решают бесконечно долго, потому что слишком глубоко анализируют.

Держим жадные алгоритмы под контролем

Когда вы сталкиваетесь с новой сложной задачей, нетрудно придумать жадное решение, используя четыре шага, описанных в предыдущем блоке. Все, что вам нужно сделать, — это разделить задачу на фазы и определить, какое жадное правило применять на каждом шаге. То есть вы делаете следующее:

- » выбираете способ принятия решения (определяете, какой подход самый простой, интуитивно понятный, минимальный и быстрый);
- » начинаете решать задачу, применяя выбранное правило принятия решений;
- » записываете результат своего решения (если необходимо) и определяете статус задачи;
- » повторно применяете тот же подход на каждом шаге до достижения окончательного решения.

Независимо от того, как вы применяете предыдущие шаги, нужно определить, достигаете ли вы своей цели, полагаясь на серию близоруких решений. Жадный подход работает для некоторых задач, но не для всех. Например, задача размена денег прекрасно работает с валютой США, но дает неоптимальные результаты с другими валютами. Чтобы увидеть эту проблему, рассмотрим вымышленную валюту (кредиты) с номиналами в 1, 15 и 25 кредитов. Предыдущий алгоритм не сможет выдать оптимальную сдачу для суммы в 30 кредитов:

```
print ('Change: %s (using %i bills)'
      % (change(30, [25, 15, 1])))
```

```
Change: [25, 1, 1, 1, 1, 1] (using 6 bills)
```


Очевидно, что оптимальное решение — вернуть две купюры по 15 кредитов, но алгоритм, будучи близоруким, начал с наибольшего доступного номинала (25 кредитов), а затем использовал пять купюр по одному кредиту для компенсации остаточных пяти кредитов.



ПРИМЕЧАНИЕ

Некоторые сложные математические структуры, называемые *матроидами* (подробнее см. на <https://jeremykun.com/2014/08/26/when-greedy-algorithms-are-perfect-the-matroid/>), могут помочь проверить, можно ли использовать жадное решение для оптимального решения конкретной задачи. Если задачу можно сформулировать с помощью матроидной структуры, жадное решение даст оптимальный результат. Тем не менее существуют задачи, имеющие оптимальные жадные решения, которые не соответствуют матроидной структуре. (О том, что матроидные структуры являются достаточными, но не необходимыми для оптимального жадного решения, можно прочитать в статье по адресу <http://csttheory.stackexchange.com/questions/21367/does-every-greedy-algorithm-have-matroid-structure>.)

Пользователь жадных алгоритмов должен знать, что они хорошо работают, но не всегда дают наилучшие возможные результаты. Когда они это делают, это происходит потому, что задача состоит из известных примеров, или потому, что задача совместима с математической структурой матроидов. Даже когда жадный алгоритм лучше всего работает в одних условиях, изменение условий может «сломать игрушку» и привести лишь к хорошим или приемлемым решениям. На самом деле, случаев просто хороших или приемлемых результатов много, поскольку жадные алгоритмы часто не превосходят другие решения, как показано здесь:

- » Рассмотренные ранее решения задачи размена денег демонстрируют, как изменение условий может привести к тому, что жадный алгоритм перестает работать.
- » Задача планирования (описанная в параграфе «Узнайте, как жадность может быть полезной» далее в этой главе) иллюстрирует, как жадное решение отлично работает с одним исполнителем, но не с несколькими.
- » Алгоритм Дейкстры для поиска кратчайшего пути работает только с ребрами, имеющими положительные веса (отрицательные веса приведут к тому, что алгоритм будет бесконечно зацикливаться вокруг некоторых узлов).

Доказать, что жадный алгоритм работает лучше всего, — сложная задача, требующая основательных математических знаний. В качестве альтернативы можно построить доказательство более эмпирическим путем, сравнивая жадное решение с одним из следующих подходов:

- » Один из методов, используемых для доказательства корректности жадных алгоритмов, – это метод *жадного обмена*. Доказательство методом жадного обмена преобразует решение, полученное любым другим алгоритмом, в решение, созданное вашим жадным алгоритмом, таким образом, что качество решения не ухудшается. Использование этого метода показывает, что любое другое решение не лучше жадного решения, что доказывает оптимальность последнего.
- » Другой алгоритм – когда по мере развертывания жадного решения вы замечаете, что оно *опережает* оптимальный алгоритм, то есть жадное решение на каждом шаге всегда обеспечивает результат лучше, чем другой алгоритм.



СОВЕТ

Даже учитывая, что успешное применение жадного подхода для нахождения оптимального решения — это скорее исключение, чем правило, жадные решения часто превосходят другие пробные решения. Вы не всегда получите наилучшее решение, но результат будет достаточно хорошим, чтобы служить как минимум отправной точкой, поэтому при решении новых задач стоит начинать именно с жадных решений.

Рассмотрение NP-полных задач

Обычно о жадном алгоритме задумываются, когда другие варианты не могут вычислить нужное решение за приемлемое время. Жадный подход подходит для задач, где у вас много вариантов выбора, которые нужно комбинировать. По мере увеличения числа возможных комбинаций сложность взрывается и даже самый мощный из доступных компьютеров не может дать ответ за разумное время. Например, пытаясь решить головоломку, можно попробовать ее решить, определив все возможные способы соединения имеющихся частей. Более разумный способ — начать решение, выбрав одну позицию, а затем найти наиболее подходящую для нее деталь. Решение головоломки таким способом означает, что вы тратите время на поиск наилучшей детали, но вам не нужно снова рассматривать эту позицию, уменьшая общее количество деталей для каждой итерации.

Головоломки, в которых количество возможных решений может стать огромным, встречаются чаще, чем можно ожидать. Некоторые задачи такого типа уже решены, но многие другие — нет, и их (пока) невозможно преобразовать в версии с известными решениями. Пока кто-нибудь не окажется достаточно умным, чтобы найти универсальное решение для этих проблем, жадный подход может быть самым простым способом их решения. При этом нужно принять тот факт, что вы не всегда получите оптимальное решение, а лишь приемлемое (во многих случаях).

Эти сложные задачи различаются по своим характеристикам и предметным областям. Один из примеров сложной задачи — сворачивание белков (что может помочь в лечении рака и недавно достигло значительного прогресса благодаря Google DeepMind и его системе искусственного интеллекта на основе глубокого обучения под названием AlphaFold2: <https://www.nature.com/articles/d41586-020-03348-4>). Другой пример сложной задачи — взлом надежного шифрования паролей, такого как популярная криптосистема RSA (<https://www.okta.com/identity-101/rsa-encryption>). В 1960-х годах исследователи обнаружили общую закономерность для всех методов надежного шифрования паролей: они все одинаково сложны для взлома. Эта закономерность получила название теории *NP-полноты* («NP» означает «недетерминированный полиномиальный»). В некотором смысле сложные задачи отличаются от других тем, что для них пока невозможно найти решение за разумное время, то есть за полиномиальное время.

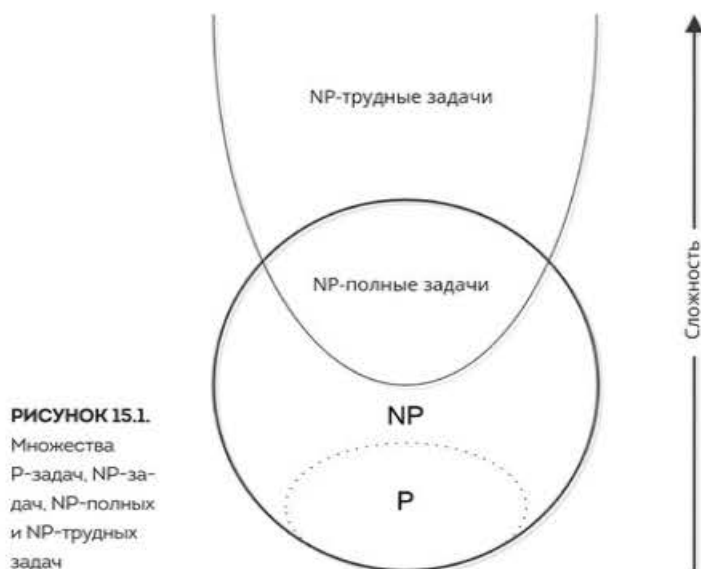
Полиномиальное время означает, что в худшем случае алгоритм выполняется за время, пропорциональное степени количества входных данных (известные как *P-задачи*). Линейное время является полиномиальным, поскольку имеет сложность $O(n^1)$. Квадратичная $O(n^2)$ и кубическая $O(n^3)$ сложности тоже полиномиальные, и хотя они растут довольно быстро, их нельзя сравнить с экспоненциальным временем $O(c^n)$. *Экспоненциальная временная сложность* делает невозможным нахождение разумного решения для любой из этих задач методом полного перебора. Фактически если n достаточно велико, вам может потребоваться перебрать количество решений, превышающее число атомов в известной Вселенной. Эта проблема входит в область NP-задач, и в настоящее время вы можете надеяться решить любую из них за полиномиальное время, только если у вас есть недетерминированная машина Тьюринга. Такая машина представляет собой своего рода магическое устройство (поэтому оно существует только в теории), которое потенциально могло бы работать благодаря бесконечному количеству параллельных процессоров. Каждый параллельный поток вычислений, хотя и не взаимодействует с другими, может обрабатывать часть решения, поэтому теоретически в итоге он может решать сложные задачи за полиномиальное время, тем самым избегая ловушки экспоненциального времени.

Поскольку недетерминированные машины Тьюринга лишь теоретические, быстрого способа решения NP-задач пока не существует. Однако эти задачи все равно решаются, потому что люди достаточно умны, чтобы находить решения. Более того, когда мы находим решение NP-задачи, легко проверить его правильность, поскольку по определению проверка решения занимает только полиномиальное время. Например, линейное программирование долгое время считалось NP-задачей, но в 1979 году российский математик Леонид Хачиян доказал с помощью алгоритма

эллипсоидов, что это P-задача. Поэтому не стоит отчаиваться — рано или поздно многие NP-задачи становятся P-задачами благодаря усилиям исследователей и ученых.

Среди NP-задач существует подмножество, называемое *NP-полными задачами* (см. рисунок 15.1). Эти задачи особенные, поскольку мы знаем, что их решение может стать ключом к одновременному решению многих других NP-задач за полиномиальное время. Эксперты по алгоритмам надеются, что в будущем кто-нибудь найдет способ решить хотя бы одну из этих задач, тем самым открыв дверь к решению всех NP-задач одновременно. Решение NP-полных задач — одно из «задач тысячелетия», предложенных Математическим институтом Клэя, который предлагает награду в 1 миллион долларов США любому, кто сможет найти решение (<http://www.claymath.org/millennium-problems/p-vs-np-problem>).

Однако NP-полные задачи не исчерпывают все сложные задачи, которые можно решить и использовать для одновременного решения всех NP-задач, как показано на рисунке 15.1. На самом деле, NP-полные задачи — это часть гораздо большего множества задач, называемых *NP-трудными задачами*. То, что NP-полные задачи также входят в класс NP, делает их достаточно легко проверяемыми (как упоминалось ранее, их можно проверить за полиномиальное время), но NP-трудные задачи не входят в NP. Это означает, что проверка их решений также требует экспоненциального времени, что делает их решение и проверку настоящим кошмаром для современных алгоритмических возможностей. Если ученым удастся совершить прорыв в решении сложной задачи и найти святой Грааль алгоритмов, это, вероятно, произойдет при решении NP-полной, а не NP-трудной задачи.



§ 2. Как жадность может быть полезна

В предыдущих блоках главы было дано общее представление о жадных алгоритмах. В этой части главы некоторые из этих алгоритмов рассматриваются подробнее, чтобы дать вам более глубокое понимание и помочь определить, как использовать их стратегии для решения других задач. В следующих блоках описывается, как работает *компьютерный кеш* (алгоритм, который всегда присутствует в любом компьютере). Кроме того, вы узнаете, как правильно планировать задачи с учетом сроков и приоритетов. Производство материальных товаров во многом опирается на жадные алгоритмы для планирования ресурсов и деятельности. Обычно алгоритмы планирования деятельности лежат в основе программного обеспечения для планирования потребности в материалах (Material Requirements Planning, MRP) и помогают эффективно управлять производством (<http://searchmanufacturingerp.techtarget.com/definition/Material-requirements-planning-MRP>). Наконец, в следующих блоках рассматривается алгоритм кодирования Хаффмана, чтобы дать более глубокое понимание того, как он работает для создания новых эффективных систем кодирования.

Организация кешированных данных компьютера

Компьютеры часто обрабатывают одни и те же данные многократно, из-за чего некоторые люди считают их одержимыми, хотя на самом деле они просто тщательны. Получение данных с диска или из интернета требует достаточной пропускной способности и вычислительного времени. Следовательно, полезно хранить часто используемые данные в локальном хранилище, где к ним легче получить доступ (и где, возможно, они уже предварительно обработаны). *Кеш*, который обычно представляет собой серию ячеек памяти или пространство на диске, зарезервированное для этой цели, выполняет эту задачу.

Например, просматривая историю вашего веб-браузера, вы, вероятно, замечаете, что только часть трафика составляют новые веб-сайты, тогда как большую часть времени вы проводите на хорошо знакомых сайтах, запрашивая знакомые страницы. Хранение в кеше некоторых частей часто посещаемых веб-сайтов (таких, как заголовок, фон, изображения и страницы, которые редко меняются) может действительно улучшить ваш опыт работы в интернете, поскольку это уменьшает необходимость повторной загрузки данных. Вам нужны только новые данные из интернета, поскольку большая часть того, что вы хотите увидеть, уже где-то в вашем компьютере. (Кеш веб-браузера — это каталог на диске.)

Эта проблема не нова. В 1960-х годах Ласло Беладии, венгерский специалист по информатике, работавший в IBM Research, предположил, что лучший способ хранить информацию в компьютере для быстрого повторного использования — это знать, какие данные понадобятся в будущем и на какой срок. На практике такое прогнозирование реализовать невозможно, поскольку использование компьютера может быть непредсказуемым и непредопределенным.

Тем не менее, как принцип, идея предвидения будущего может вдохновить на создание оптимальной *стратегии замещения* — жадного выбора, основанного на идее сохранения страниц, которые, как ожидается, будут использоваться в ближайшее время, то есть на предыдущих запросах к кешу. Оптимальная политика замещения страниц Беладии (также известная как *ясновидящий алгоритм замещения*) работает по жадному принципу: удалять из кеша данные, следующее использование которых, вероятно, произойдет в самом отдаленном будущем, чтобы минимизировать вероятность удаления того, что может понадобиться в ближайшее время. Для реализации этой идеи алгоритм следует следующим шагам:

- 1 Заполнить компьютерный кеш, записывая данные каждого сделанного запроса.** Только когда кеш заполнен, вы начинаете удалять старые данные, чтобы освободить место для новых.
- 2 Определить метод для установления недавнего использования.** Этот алгоритм может использовать временные метки файлов или систему флагов в памяти (которая отмечает недавно использованные страницы и очищает все флаги через определенное время) для принятия решения.
- 3 Когда нужно разместить новые данные — из кеша удалить данные, которые не использовались в последнее время.** Алгоритм случайным образом выбирает один элемент данных среди неиспользуемых.

Например, если в вашем кеше есть только четыре ячейки памяти и они заполнены четырьмя буквами алфавита, которые поступили в следующем порядке:

A B C D

то, когда обрабатывается новая буква, к примеру буква *E*, компьютер освобождает для нее место, удаляя одну из букв, которые с меньшей вероятностью будут сейчас запрошены. В этом примере хорошими кандидатами являются *A*, *B* или *C* (*D* — самое недавнее добавление). Алгоритм случайным образом выберет одну ячейку и удалит из кеша содержащиеся в ней данные, чтобы освободить место для *E*.

Конкуренция за ресурсы

Нет, этот блок не о новом игровом шоу, где победители уходят с потрясающими призами. Вместо этого речь идет о решении распространенной проблемы, которая возникает, когда вы хотите достичь определенной цели, например создать услугу или произвести материальный объект. Проблема заключается в планировании нескольких конкурирующих действий, требующих эксклюзивного доступа к ресурсам. Ресурсы могут включать время работы на производственном оборудовании. Примеров таких ситуаций в реальном мире много: от составления расписания посещения университетских курсов до организации армейских поставок, от сборки сложного продукта, такого как автомобиль, до организации последовательности вычислительных задач в центре обработки данных. Неизменно общими целями в таких ситуациях являются:

- » выполнение максимального количества задач за определенное время;
- » управление задачами максимально быстро в среднем;
- » соблюдение строгих приоритетов (жесткие сроки);
- » соблюдение указаний по приоритетам (мягкие сроки).

Планирование задач делится на две категории:

- » задачи, которые сложно решить правильно и которые требуют использования продвинутых алгоритмов;
- » задачи, с которыми легче справиться и которые можно решить с помощью простых жадных алгоритмов.

Большинство задач планирования, которые вы выполняете, фактически относятся к тем, что решаются жадными алгоритмами. Например, управление задачами максимально быстро — это обычное требование для промышленного производства или сферы услуг, когда каждая задача обслуживает потребности клиента и вы хотите сделать все возможное для всех своих клиентов. Вот как можно определить контекст для такого алгоритма:

- » у вас есть одна машина (или работник), которая может выполнять заказы;
- » заказы поступают партиями, поэтому у вас есть много вариантов для выбора одновременно;
- » заказы различаются по длительности, каждый требует разного времени выполнения.

Например, вы получили четыре задания от четырех бизнес-клиентов, на выполнение которых требуется соответственно 8, 4, 12 и 3 часа. Хотя общее время выполнения остается неизменным, перемена порядка выполнения заданий влияет на время их завершения и определяет, как долго каждому бизнес-клиенту придется ждать выполнения своего заказа. В следующих блоках рассматриваются различные методы удовлетворения потребностей бизнес-клиентов с учетом конкретных целей.

Обеспечение удовлетворенности клиентов

Бизнес строится на том, чтобы клиенты были довольны. Если выполнять задания в представленном порядке, работа займет $8 + 4 + 12 + 3 = 27$ часов. При этом первый клиент получит свой заказ через 8 часов, а последний — через 27 часов. Фактически первое задание будет выполнено через 8 часов, второе — через $8 + 4 = 12$ часов, третье — через $8 + 4 + 12 = 24$ часа, последнее — через $8 + 4 + 12 + 3 = 27$ часов.

Если ваша цель — сделать всех клиентов довольными и удовлетворенными, следует стремиться минимизировать среднее время ожидания для каждого из них. Этот показатель определяется средним значением времени выполнения: $(8 + 12 + 24 + 27) / 4 = 17,75$ часа в среднем на ожидание выполнения задания. Чтобы сократить среднее время ожидания, можно начать моделировать все возможные комбинации порядка выполнения и пересчитывать оценку. Это выполнимо для нескольких заданий на одной машине; но если у вас их сотни на нескольких машинах, это становится очень сложной вычислительной задачей. Жадный алгоритм может спасти положение без особого планирования — просто выполняйте сначала самые короткие задания. Полученное среднее значение будет минимально возможным: $(3 + (3 + 4) + (3 + 4 + 8) + (3 + 4 + 8 + 12)) / 4 = 13$ часов в среднем.



ЗАПОМНИТЕ

Чтобы получить среднее время ожидания, вы берете среднее значение накопленных сумм времени выполнения. Если вместо этого взять среднее значение исходного времени, вы получите среднюю продолжительность задания, которая не отражает времени ожидания клиента.

Принцип жадного алгоритма прост: поскольку вы суммируете накопительные времена, то, если начать с выполнения самых длительных заданий, вы увеличиваете самое длительное время выполнения для всех последующих операций (поскольку это накопительная сумма). Если же начать с самых коротких заданий, вы сначала получаете наименьшее время, что положительно влияет на среднее значение (и на средний уровень удовлетворенности ваших клиентов, хотя клиент с самым длительным заданием все равно может остаться недовольным).

Соблюдение сроков

Иногда, помимо желания сократить время ожидания клиентов, вам также нужно соблюдать их временные требования, то есть учитывать крайние сроки. Когда у вас есть дедлайны, механизм жадного алгоритма меняется. Теперь вы начинаете не с самого короткого задания, а с того, которое нужно выполнить раньше, следуя принципу *«чем раньше, тем лучше»*. Это проблема жестких дедлайнов, и в ее решении можно потерпеть неудачу. (Некоторые сроки просто невозможно соблюсти.)



СОВЕТ

Если вы попробуете жадную стратегию и у вас не получится решить проблему, вы можете признать, что решения для требуемого срока не существует. Когда жесткие дедлайны не работают, можно попытаться решить проблему с помощью мягких дедлайнов, что означает необходимость соблюдения приоритетности (выполнение определенных заданий в первую очередь).

В этом примере у вас есть как длительности задач, рассмотренные в предыдущем блоке, так и значения (*веса*), определяющие важность задачи (у больших весов более высокий приоритет). Это та же задача, но теперь вам нужно минимизировать средневзвешенное время выполнения. Для достижения этой цели вы создаете приоритетный рейтинг, разделив длительности на веса, и начинаете с задач, имеющих наименьший рейтинг. Если у задачи самый низкий рейтинг, это означает, что она либо высокоприоритетная, либо очень короткая.

Возвращаясь к предыдущему примеру, допустим, что теперь у вас есть кортежи, содержащие и веса, и длительности: (40, 8), (30, 4), (20, 12), (10, 3), где 40 в первом кортеже — это вес, а 8 — длительность. Разделив каждую длительность на вес, вы получите приоритетные рейтинги: 0,20; 0,13; 0,60; 0,30. Начиная с наименьшего приоритетного рейтинга и добавляя следующий наименьший, вы получаете оптимальное расписание, которое обеспечивает как минимизацию времени, так и соблюдение приоритетов: (30, 4), (40, 8), (10, 3), (20, 12).

Возвращаясь к кодированию Хаффмана

Как было показано в предыдущей главе, кодирование Хаффмана позволяет представить данные в более компактной форме, используя тот факт, что некоторые данные (например, определенные буквы алфавита) встречаются в потоке данных чаще других. С использованием кодировки разной длины (более короткие — для наиболее частых символов, более длинные — для наименее частых) данные занимают меньше места. Профессор Роберт М. Фано (преподаватель Хаффмана) и Клод Шеннон уже предвидели такую стратегию сжатия, но не смогли найти эффективный способ определения схемы кодирования, которая исключала бы возможность принять один символ за другой.



ЗАПОМНИТЕ

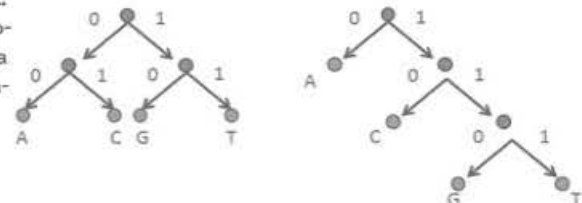
Беспрефиксные коды необходимы для предотвращения ошибок при декодировании сообщения. Это означает, что ни одна ранее использованная битовая кодировка не должна использоваться как начальная точка для другой битовой кодировки. Хаффман нашел простое и работоспособное решение для реализации беспрефиксных кодов с помощью жадного алгоритма. Решение проблемы беспрефиксности, найденное Хаффманом, заключается в преобразовании исходного сбалансированного дерева (структуры данных, рассмотренной в главе 6), содержащего кодировку фиксированной длины, в несбалансированное дерево, как показано на рисунке 15.2.

Несбалансированное дерево обладает особой характеристикой: у каждого узла есть только одна ветвь, которая продолжает развиваться в другие узлы и ветви, в то время как другая ветвь заканчивается закодированным символом. Эта особенность гарантирует, что ни одна ранее использованная последовательность кодирования не может начать новую последовательность (графически ветвь, заканчивающаяся закодированным символом, представляет собой тупик).

Помимо графического построения несбалансированной структуры, жадный алгоритм также может создать несбалансированное дерево. Идея заключается в построении структуры сверху вниз от корня, начиная с наименее часто используемых символов. Алгоритм создает нижние уровни дерева путем последовательного объединения менее частых символов, пока не закончатся все символы и не будет достигнуто дно.

РИСУНОК 15.2.

От сбалансированного дерева (слева) к несбалансированному дереву (справа)



Чтобы продемонстрировать жадный принцип, лежащий в основе алгоритма, в этом блоке приводится пример кода на Python, основанный на ДНК. ДНК представлена как последовательность букв *A*, *C*, *T* и *G* (четыре нуклеотида, присутствующих во всех живых существах). Хороший прием — использовать всего 2 бита для представления каждой из четырех букв, что уже является эффективной стратегией экономии памяти в сравнении с использованием полной кодировки ASCII (которая составляет как минимум 7 бит).

Нуклеотиды распределены неравномерно. Распределение варьируется в зависимости от изучаемых генов. В следующей таблице показан ген с неравномерным распределением, где преобладают нуклеотиды *A* и *C*.

Нуклеотиды	Процент, %	Фиксированное кодирование	Кодирование Хаффмана
A	40,5	00	0
C	29,2	01	10
G	14,5	10	110
T	15,8	11	111
Средневзвешенное количество битов:		2,00	1,90

Умножая количество битов двух кодировок на их процентное соотношение и суммируя всё, вы получаете средневзвешенное значение битов, используемых каждой из них. В данном случае результат составляет 1,9 для кодирования Хаффмана против 2,0 для фиксированного кодирования. Это означает, что в данном примере вы получаете 5-процентную экономию битов. Можно сэкономить еще больше места при работе с генами, имеющими еще более несбалансированное распределение в пользу какого-либо нуклеотида.

Следующий пример генерирует случайную последовательность ДНК и показывает, как код систематически генерирует кодировку. (Если вы измените начальное значение, случайная генерация последовательностей ДНК может привести к другому результату как в распределении нуклеотидов, так и в кодировании Хаффмана.)

```

from random import shuffle, seed
from collections import defaultdict, Counter

generator = ["A"]*6+["C"]*4+["G"]*2+["T"]*2
text = ""
seed(4)
for i in range(1000):
    shuffle(generator)
    text += generator[0]

frequencies = Counter(list(text))
print(f"{text[:25]} ... \n{frequencies}")

CAACCCCGACACGCCTCCATAGCCA ...
Counter({'A': 405, 'C': 292, 'T': 158, 'G': 145})

```

Подготовив входные данные для сжатия, код создает структуру данных в виде кучи (подробнее см. в блоке «Выполнение специализированного поиска с помощью бинарной кучи» главы 7) для эффективного упорядочивания результатов на каждом шаге алгоритма. Элементы в куче содержат символы нуклеотидов и частоту их встречаемости. С логарифмической временной сложностью $O(n \cdot \log(n))$ куча является подходящей структурой для упорядочивания результатов и позволяет алгоритму быстро извлекать два наименьших элемента `quickly.class BinaryHeap()`:

```

def __init__(self):
    self.heap = []

def swap(self, i, j):
    self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

def insert(self, key, value):
    index = len(self.heap)
    self.heap.append([key, value])
    while index != 0:
        parent = (index - 1) // 2
        if self.heap[parent][1] < self.heap[index][1]:
            self.swap(parent, index)
        index = parent

heap = BinaryHeap()
for key_value_pair in frequencies.items():
    heap.insert(*key_value_pair)
print(heap.heap)

[['A', 405], ['C', 292], ['G', 145], ['T', 158]]

```


Когда вы запускаете алгоритм, основываясь на сортировке, выполненной бинарной кучей, он выбирает нуклеотиды с меньшей частотой из кучи (жадный выбор). Затем он объединяет эти нуклеотиды в новый элемент, заменяя два предыдущих. Процесс продолжается до тех пор, пока последнее объединение не уменьшит количество оставшихся для извлечения из кучи элементов до одного.

```
encoding = {item[0]:'' for item in heap.heap}
for i in range(1, len(heap.heap)):
    aggregate = heap.heap[-i:]
    new = heap.heap[-i-1]
    # putting a 1 in front of previous element
    for item in aggregate:
        encoding[item[0]] = '1' + encoding[item[0]]
    # putting a 0 in front of following element
    encoding[new[0]] = '0' + encoding[new[0]]
    print(f"{aggregate} + {new} =\n{encoding}\n")
```

```
['T', 158] + ['G', 145] =
{'A': '', 'C': '', 'G': '0', 'T': '1'}
['G', 145], ['T', 158] + ['C', 292] =
{'A': '', 'C': '0', 'G': '10', 'T': '11'}
['C', 292], ['G', 145], ['T', 158] + ['A', 405] =
{'A': '0', 'C': '10', 'G': '110', 'T': '111'}
```

По мере того как агрегации объединяют нуклеотиды, формируя различные уровни несбалансированного дерева, их кодировка по Хаффману систематически модифицируется: добавляется ноль перед кодировкой элемента, который должен войти в агрегат, и единица — перед кодировками элементов, уже находящихся в агрегате. Таким способом алгоритм эффективно воспроизводит структуру несбалансированного дерева, показанную ранее.

Заключительный шаг — вывод результата, демонстрирующий, как формировался агрегат и как менялись кодировки. Последняя агрегация представляет собой итоговую сгенерированную таблицу символов.

В ЭТОЙ ГЛАВЕ

- » Понимание того, что означает «динамический», когда этот термин используется в программировании
- » Эффективное использование мемоизации для динамического программирования
- » Изучение того, как задача о рюкзаке может быть полезна для оптимизации
- » Работа с NP-полной задачей коммивояжера

ГЛАВА 16

Применение динамического программирования

Вместо использования полного перебора, подразумевающего проверку всех возможных решений задачи, жадные алгоритмы предоставляют быстрый и зачастую удовлетворительный ответ. В некоторых случаях жадный алгоритм может даже полностью решить проблему. Однако жадные алгоритмы также ограничены тем, что принимают решения, не учитывая последствия своего выбора. В главе 15 показано, что не всегда возможно решить задачу с помощью жадного алгоритма.

В этой главе представлено динамическое программирование. Объясняется, как алгоритм может принять решение, кажущееся оптимальным на определенном этапе, но позже оказывающееся ограничивающим и неоптимальным для достижения наилучшего решения. Более совершенный алгоритм, не опирающийся на жадный подход, может пересматривать прошлые решения или предвидеть, что внешне хорошее решение не так перспективно, как может показаться. Именно такой подход использует динамическое программирование.

Эта глава предлагает больше, чем просто определение динамического программирования. В первом параграфе также объясняется, почему у динамического программирования такое сложное название. Кроме того, вы узнаете, как преобразовать любой алгоритм (особенно рекурсивный) в динамическое программирование с помощью Python и его *декораторов функций* (мощных инструментов Python, позволяющих изменять существующую функцию без переписывания ее кода). Практические примеры объясняют динамическое программирование лучше, чем теория, поэтому второй параграф главы показывает применение динамического программирования для оптимизации ресурсов и доходов, создания коротких маршрутов между пунктами и приближенного сравнения строк. Динамическое программирование предоставляет естественный подход к решению многих задач, с которыми вы сталкиваетесь, путешествуя по миру алгоритмов.



ЗАПОМНИТЕ

Вам не нужно вручную набирать исходный код для этой главы. На самом деле, использовать загружаемый исходный код намного проще. Исходный код главы можно найти в файлах `\A4D2E\A4D2E; 16; Fibonacci.ipynb`, `\A4D2E\A4D2E; 16; Knapsack.ipynb`, `\A4D2E\A4D2E; 16; Levenshtein.ipynb` и `\A4D2E\A4D2E; 16; TSP.ipynb` из загружаемых материалов. Подробнее о том, как найти эти файлы, смотрите во введении.

§ 1. Объяснение динамического программирования

Динамическое программирование — это алгоритмический подход, разработанный в 1950-х годах Ричардом Эрнестом Беллманом (прикладным математиком, известным также другими открытиями в области математики и алгоритмов; подробнее можно прочитать на <https://mathshistory.st-andrews.ac.uk/Biographies/Bellman>), который проверяет больше решений, чем соответствующий жадный подход. Проверка большего количества решений дает возможность переосмыслить и оценить последствия принятых решений. Динамическое программирование избегает выполнения сложных вычислений благодаря умной системе кеширования, называемой *мемоизацией* , термин которой будет описан позже в главе. (*Кеш* — это система хранения, которая собирает данные или информацию.)

Динамическое программирование столь же эффективно, как и полный перебор, — таким образом оно предоставляет правильные решения, — но при этом часто так же эффективно, как и приближенное решение (время вычисления многих алгоритмов динамического программирования поли-

номиально). Кажется, что оно работает как магия, потому что нужное вам решение часто требует от алгоритма многократного выполнения одних и тех же вычислений. Модифицируя алгоритм и делая его динамическим, вы можете записывать результаты вычислений и повторно использовать их при необходимости. Повторное использование занимает мало времени в сравнении с пересчетом, поэтому алгоритм быстро завершает все шаги. В следующих блоках подробнее рассматривается, что включает динамическое программирование.

Исторический экскурс

Динамическое программирование можно свести к тому, что алгоритм запоминает результаты предыдущих задач, где в противном случае пришлось бы выполнять одни и те же вычисления многократно. Хотя динамическое программирование может показаться довольно сложным, его реализация на самом деле достаточно проста. Однако у него есть интересные исторические корни.

Беллман описал название *динамическое программирование* как результат необходимости и удобства в своей автобиографии «*В глазу урагана*». Он пишет, что выбор названия был способом скрыть истинную природу его исследований в корпорации RAND (научно-исследовательский институт, финансируемый как правительством США, так и частными инвесторами) от Чарльза Эрвина Уилсона, министра обороны при президенте Эйзенхауэре. Маскировка истинной природы исследований помогла Беллману сохранить работу в корпорации RAND. Более подробно его объяснение можно прочесть в отрывке по адресу http://theory.cs.utah.edu/fall18/algorithms/dy_birth.pdf. Некоторые исследователи не согласны с происхождением названия. Например, Стюарт Рассел и Питер Норвиг в книге «*Искусственный интеллект: современный подход*» утверждают, что Беллман использовал термин *динамическое программирование* еще в статье 1952 года, то есть до того, как Уилсон стал министром в 1953 году (к тому же сам Уилсон был генеральным директором General Motors, прежде чем стать инженером, занимающимся исследованиями и разработками).

Языки программирования не были широко распространены в то время, когда Беллман работал в области исследования операций — дисциплине, применяющей математику для принятия более эффективных решений в основном при решении производственных или логистических задач (хотя она используется и для других практических проблем). Вычислительная техника находилась на ранних стадиях развития и использовалась в основном для планирования. Базовый подход динамического программирования схож с *линейным программированием* — другой алгоритмической техникой (см. главу 19), определенной в те годы, когда

программирование означало планирование конкретного процесса для поиска оптимального решения. Термин *динамический* напоминает о том, что алгоритм перемещает и хранит частичные решения. Динамическое программирование — это сложное название для умной и эффективной техники улучшения времени выполнения алгоритмов.

Как сделать задачи динамическими

Поскольку динамическое программирование использует преимущества повторяющихся операций, оно хорошо работает с задачами, чьи решения строятся вокруг решения подзадач, которые алгоритм позже объединяет для получения полного ответа. Для эффективной работы подход динамического программирования использует подзадачи, вложенные в другие подзадачи. (Этот подход схож с жадными алгоритмами, которые также требуют оптимальной подструктуры, как объясняется в главе 15.) Только когда вы можете разбить задачу на вложенные подзадачи, динамическое программирование может превзойти методы полного перебора, которые постоянно перерешают одни и те же подзадачи.



ЗАПОМНИТЕ

Как концепция, динамическое программирование — это огромный зонтик, охватывающий множество различных приложений, поскольку это не конкретный алгоритм для решения определенной задачи. Скорее, это общая техника, поддерживающая решение задач. Динамическое программирование можно разделить на два больших семейства решений:

- » **восходящий подход** — строит массив частичных результатов, которые объединяются в полное решение;
- » **нисходящий подход** — разбивает задачу на подзадачи, начиная с полного решения (этот подход типичен для рекурсивных алгоритмов), и использует мемоизацию (описывается в следующем блоке) для избежания повторных вычислений.

Как правило, нисходящий подход эффективнее с точки зрения вычислений, поскольку генерирует только подзадачи, необходимые для полного решения. Восходящий подход более исследовательский и, используя метод проб и ошибок, часто получает частичные результаты, которые не будут использованы позже. С другой стороны, восходящие подходы лучше отражают тот подход, который мы используем в повседневной жизни при решении проблем (рекурсивное мышление, напротив, требует абстракции и тренировки перед применением). Оба подхода — и восходящий, и нисходящий — порой нелегко понять. Это потому, что использование динамического программирования трансформирует способ решения задач, как описано в следующих шагах:

- 1 Создание рабочего решения с использованием полного перебора и, возможно, рекурсии. Решение работает, но занимает много времени или вообще не завершается.
- 2 Сохранение результатов подзадач, чтобы ускорить вычисления и получить решение за разумное время.
- 3 Изменение подхода к решению задачи для достижения еще большей скорости.
- 4 Переосмысление подхода к задаче менее интуитивным, но более эффективным способом, чтобы извлечь максимальную выгоду из динамического программирования.

Преобразование алгоритмов с помощью динамического программирования для повышения их эффективности обычно делает их сложнее для понимания. Фактически глядя на такие решения, можно подумать, что они работают как по волшебству. Чтобы овладеть динамическим программированием, требуется как многократное изучение существующих решений, так и практические упражнения. Однако эти усилия того стоят, поскольку динамическое программирование помогает решать задачи, требующие систематического сравнения или вычисления решений.



СОВЕТ

Динамическое программирование особенно известно тем, что помогает решать (или, по крайней мере, делать менее затратными по времени) задачи комбинаторной оптимизации — задачи, требующие получения комбинаций входных элементов в качестве решения. Примеры таких задач, решаемых методом динамического программирования, — задача коммивояжера и задача о рюкзаке, которые описаны далее в главе.

Преобразование рекурсии в динамическое решение

Суть динамического программирования заключается в том, чтобы достичь такой же эффективности, как при полном переборе, но без фактических затрат времени на вычисления, требуемые при подходе полного перебора. Этот результат достигается путем обмена времени на дисковое пространство или память, что обычно делается посредством создания структуры данных (хеш-таблицы, массива или матрицы данных) для хранения предыдущих результатов. Использование таблиц поиска позволяет получать доступ к результатам без необходимости повторного выполнения вычислений.



ЗАПОМНИТЕ

Метод сохранения предыдущих результатов функции и их использования вместо повторного вычисления называется *мемоизацией* (не путать со словом «меморизация» — «запоминание»). Термин «мемоизация» происходит от латинского *memorandum*, что означает «то, что следует запомнить».



ЗАПОМНИТЕ

Кеширование — еще один термин, который часто используется в контексте мемоизации. *Кеширование* означает использование специальной области компьютерной памяти для более быстрого предоставления данных при необходимости и имеет более широкое применение, чем мемоизация.

Для эффективности динамического программирования нужны задачи, которые повторяют предыдущие шаги или возвращаются к ним. Хороший пример подобной ситуации — использование рекурсии, а классический пример рекурсии — это вычисление чисел Фибоначчи. Последовательность Фибоначчи — это просто последовательность чисел, в которой каждое следующее число является суммой двух предыдущих. Последовательность начинается с 0, за которым следует 1. После определения первых двух элементов каждое следующее число в последовательности является суммой предыдущих. Вот первые одиннадцать чисел:

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Как и при индексации в Python, отсчет начинается с нулевой позиции, и последнее число в последовательности находится на десятой позиции. Создатель этой последовательности, итальянский математик Леонардо Пизанский, известный как Фибоначчи, жил в 1200 году. Фибоначчи полагал: тот факт, что каждое число — это сумма двух предыдущих, должен был сделать эти числа подходящими для представления закономерностей роста популяции кроликов. Последовательность не очень хорошо подошла для демографии кроликов, но предоставила неожиданные идеи как в математике, так и в самой природе, поскольку эти числа встречаются в ботанике и зоологии. Например, эту прогрессию можно увидеть в ветвлении деревьев, в расположении листьев на стебле и семян в подсолнухе (об этом расположении можно прочитать на <https://www.goldennumber.net/spirals>).



ЗАПОМНИТЕ

Фибоначчи также был математиком, который познакомил Европу с индо-арабскими цифрами — системой, которой мы пользуемся ежедневно. Он описал как сами числа, так и последовательность в своем шедевре «Книга абака» (*Liber Abaci*) в 1202 году.

Вы можете вычислить последовательность чисел Фибоначчи с помощью рекурсии. Когда вы вводите число, рекурсия разбивает его на сумму двух предыдущих чисел Фибоначчи в последовательности. После первого разбиения рекурсия продолжает выполнять ту же задачу для каждого элемента разбиения, разделяя каждое из двух чисел на предыдущие два

числа Фибоначчи. Рекурсия продолжает разбивать числа на их суммы, пока не дойдет до корней последовательности — чисел 0 и 1. С учетом двух типов алгоритмов динамического программирования, описанных в блоке «Как сделать задачи динамическими» ранее в этой главе, данное решение использует нисходящий подход. Следующий код демонстрирует рекурсивный подход на Python. (Вы можете найти этот код в загружаемом файле исходного кода **A4D2E; 16; Fibonacci.ipynb**; подробнее см. во введении.)

```
def fib(n, tab = 0):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        print(f"lvl {tab}", end = ': ')
        print(f"summing fib({n - 1}) and fib({n - 2})")
        return fib(n - 1, tab + 1) + fib(n - 2, tab + 1)
```

Код выводит разбиения, генерируемые на каждом уровне рекурсии. Следующий вывод показывает, что происходит при вызове `fib()` со входным значением 7:

```
fib(7)

lvl 0: summing fib(6) and fib(5)
lvl 1: summing fib(5) and fib(4)
lvl 2: summing fib(4) and fib(3)
lvl 3: summing fib(3) and fib(2)
lvl 4: summing fib(2) and fib(1)
lvl 5: summing fib(1) and fib(0)
lvl 4: summing fib(1) and fib(0)
lvl 3: summing fib(2) and fib(1)
lvl 4: summing fib(1) and fib(0)
lvl 2: summing fib(3) and fib(2)
lvl 3: summing fib(2) and fib(1)
lvl 4: summing fib(1) and fib(0)
lvl 3: summing fib(1) and fib(0)
lvl 1: summing fib(4) and fib(3)
lvl 2: summing fib(3) and fib(2)
lvl 3: summing fib(2) and fib(1)
lvl 4: summing fib(1) and fib(0)
lvl 3: summing fib(1) and fib(0)
lvl 2: summing fib(2) and fib(1)
lvl 3: summing fib(1) and fib(0)
13
```

Вывод показывает 20 разбиений. Некоторые числа появляются более одного раза как часть разбиений. Похоже на идеальный случай для применения динамического программирования. Следующий код добавляет словарь под названием **memo**, который хранит предыдущие результаты. После того как рекурсия разбивает число, она проверяет, есть ли уже результат в словаре, прежде чем начинать следующую рекурсивную ветвь. Если результат найден, код использует предварительно вычисленный результат, как показано здесь:

```
memo = dict()
def fib_mem(n, tab = 0):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        if (n-1, n-2) not in memo:
            print (f"lvl {tab}", end=': ')
            print (f"summing fib({n-1}) and fib({n-2})")
            memo[(n-1,n-2)] = fib_mem(n-1,tab+1
                                     ) + fib_mem(n-2,tab+1)
        return memo[(n-1,n-2)]
```

Используя мемоизацию, рекурсивная функция выполняет не 20 сложений, а всего шесть — те основные операции, которые используются как строительные блоки для решения исходной задачи вычисления определенного числа в последовательности:

```
fib_mem(7)

lvl 0: summing fib(6) and fib(5)
lvl 1: summing fib(5) and fib(4)
lvl 2: summing fib(4) and fib(3)
lvl 3: summing fib(3) and fib(2)
lvl 4: summing fib(2) and fib(1)
lvl 5: summing fib(1) and fib(0)

13
```

Заглянув в словарь **memo**, можно найти последовательность сумм, которые определяют последовательность Фибоначчи, начиная с 1:

```
memo
{(1, 0): 1, (2, 1): 2, (3, 2): 3, (4, 3): 5, (5, 4): 8,
 (6, 5): 13}
```


Использование мемоизации

Мемоизация — это суть динамического программирования. Часто возникает необходимость использовать ее при написании собственного алгоритма. При создании функции, рекурсивной или нет, вы можете легко преобразовать ее с помощью простой команды — *декоратора*, который представляет собой специальную функцию Python, преобразующую другие функции. Чтобы увидеть, как работать с декоратором, начнем с рекурсивной функции, очищенной от операторов печати:

```
def fib(n):
    if n==0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

При использовании Jupyter Notebook можно применять встроенные магические команды, такие как `%timeit`, для измерения времени выполнения команды на вашем компьютере:

```
%timeit -n 1 -r 1 print(fib(36))

14930352
12.7 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Вывод показывает, что функции требуется около 12,7 секунды для выполнения на компьютере, использованном для этого теста (система Windows на процессоре Intel Core i3). Однако в зависимости от вашей машины выполнение функции может занять больше или меньше времени. Независимо от скорости вашего компьютера, это определенно займет несколько секунд, поскольку число Фибоначчи для 36 довольно огромное: 14930352. Тестирование той же функции для более высоких чисел Фибоначчи занимает еще больше времени.

Теперь пришло время увидеть эффект от декорирования функции. Использование функции `lru_cache()` из пакета `functools` может радикально сократить время выполнения. Эта функция доступна только при использовании Python 3. Он преобразует функцию, автоматически добавляя кеш для хранения ее результатов. Вы также можете задать размер кеша с помощью параметра `maxsize` (`lru_cache()` использует кеш с оптимальной стратегией замещения, как объясняется в главе 15). Если установить `maxsize=None`, кеш будет использовать всю доступную память без ограничений.

```
from functools import lru_cache

@lru_cache(maxsize=None)
def fib(n):
    if n==0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Обратите внимание, что сама функция осталась прежней. Единственное дополнение — это импортированная функция `lru_cache()` (<https://docs.python.org/3/library/functools.html>), которая вызывается путем добавления символа `@` перед ней. Любой вызов с символом `@` впереди — это *аннотация*, и в данном случае вызывает функцию `lru_cache()` как декоратор для следующей функции.



ПРИМЕЧАНИЕ

Использование декораторов — это продвинутая техника в Python. Подробное объяснение декораторов выходит за рамки книги, но вы все равно можете использовать их преимущества, поскольку они очень просты в применении. (Дополнительную информацию о декораторах можно найти на <https://pythonbasics.org/decorators> и <https://www.freecodecamp.org/news/python-decorators-explained-with-examples/>.) Просто помните, что они вызываются с помощью аннотаций (`@ + имя функции-декоратора`) и размещаются перед функцией, которую вы хотите преобразовать. Исходная функция передается в декоратор и выходит преобразованной. В этом примере простой рекурсивной функции декоратор выдает функцию рекурсии, обогащенную мемоизацией.

Пришло время протестировать скорость функции, как и раньше:

```
%timeit -n 1 -r 1 print(fib(36))

14930352
707 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Даже если время выполнения у вас будет другим, оно должно уменьшиться с секунд до миллисекунд. Такова сила мемоизации. Вы также можете исследовать, как ваша функция использует свой кеш, вызвав метод `cache_info()` у декорированной функции:

```
fib.cache_info()

CacheInfo(hits=34, misses=37, maxsize=None, currsize=37)
```

Вывод показывает, что было 37 вызовов функции, которые не нашли ответа в кеше. Однако 34 других вызова нашли полезный ответ в кеше.



СОВЕТ

Просто импортируя `lru_cache()` из `functools` и используя его в аннотациях перед многими ресурсоемкими алгоритмами в Python, вы получите значительное увеличение производительности (если только это не жадные алгоритмы).

Если вместо использования готовой функции вы хотите написать собственную функцию мемоизации, можете воспользоваться следующей функцией:

```
def memoize(fun):
    cache = dict()

    def memoized(*args):
        if args in cache:
            return cache[args]
        result = fun(*args)
        cache[args] = result
        return result

    return memoized
memoized_fib = memoize(fib)

%timeit -n 1 -r 1 print(memoized_fib(36))

14930352
956 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

В Python, когда функция возвращает другую функцию, возвращаемая функция получает доступ ко всем переменным родительской функции. Таким образом, вы можете дополнить функцию `fib()` словарем, который будет хранить все предыдущие результаты, и обращаться к ним, когда функция снова вызывается с теми же параметрами.

§ 2. Открываем лучшие динамические рецепты

Однако даже у динамического программирования есть ограничения. Самое большое ограничение связано с его главным преимуществом: если вы храните слишком много частичных решений для улучшения времени выполнения, то может закончиться память. Большое количество частичных решений может накопиться либо из-за сложности задачи, либо просто потому, что выбранный порядок генерации частичных решений неоптимален и многие решения не соответствуют требованиям задачи.

Порядок решения подзадач — это то, за чем нужно следить. Выбранный порядок должен быть логичным для эффективного продвижения алгоритма (вы решаете то, что собираетесь сразу же использовать повторно), поскольку суть заключается в разумном повторном использовании ранее созданных строительных блоков. Поэтому одной мемоизации может быть недостаточно. Перестановка задач в правильном порядке может улучшить результаты. Научиться правильно упорядочивать подзадачи можно на лучших примерах динамического программирования: задаче о рюкзаке, задаче коммивояжера и приближенном поиске строк, которые описаны в следующих блоках.

Заглядываем в рюкзак

Задача о рюкзаке существует как минимум с 1897 года и, вероятно, является работой Тобиаса Данцига (<https://www.britannica.com/biography/Tobias-Dantzig>). В этой задаче нужно наполнить рюкзак как можно большим количеством предметов. У каждого предмета своя ценность, поэтому вы хотите максимизировать общую ценность переносимых предметов. У рюкзака есть предельная вместимость, или у вас есть ограничение по весу, который вы можете нести, поэтому вы не можете взять все предметы.

Эта общая ситуация подходит для любой задачи, связанной с бюджетом и ресурсами, которые вы хотите распределить наиболее разумным образом. Задача о рюкзаке настолько распространена, что многие считают ее одной из самых популярных алгоритмических задач. Она находит применение в информатике, производстве, финансах, логистике и криптографии. Например, в реальном мире задача о рюкзаке помогает оптимально загрузить грузовое судно товарами или наилучшим образом раскрыть сырье, чтобы минимизировать отходы.



ЗАПОМНИТЕ

Несмотря на такую популярность задачи о рюкзаке, в книге мы не будем повторно ее рассматривать, поскольку динамический подход, бесспорно, один из лучших методов ее решения. Важно помнить, однако, что в определенных случаях — например, когда речь идет о количествах — другие подходы, такие как жадные алгоритмы, могут работать не хуже (или даже лучше).

В этом блоке показано, как решить *задачу о рюкзаке 0–1*. В этом варианте у вас есть конечное число предметов и каждый из них можно либо положить в рюкзак (статус «один»), либо нет (статус «ноль»). Полезно знать, что существуют и другие варианты этой задачи:

- » **задача о дробном рюкзаке** — имеет дело с количествами, например предметом могут быть килограммы муки и нужно выбрать оптимальное количество (этот вариант можно решить с помощью жадного алгоритма);
- » **задача об ограниченном рюкзаке** — позволяет положить одну или несколько копий одного и того же предмета в рюкзак (в этом случае нужно учитывать минимальные и максимальные требования к количеству для каждого выбранного предмета);
- » **задача о неограниченном рюкзаке** — позволяет положить одну или несколько копий одного и того же предмета в рюкзак без ограничений (единственное ограничение — нельзя положить отрицательное количество предметов в рюкзак).

Задача о рюкзаке 0–1 решается методом динамического программирования и выполняется за псевдополиномиальное время (что хуже, чем просто полиномиальное время), поскольку время выполнения зависит от количества предметов (n), умноженного на число долей вместимости рюкзака (W), которые используются при построении частичного решения. В нотации Big-O можно сказать, что время выполнения составляет $O(nW)$. Версия алгоритма с полным перебором, напротив, выполняется за $O(2^n)$. Алгоритм работает следующим образом:

- 1 При заданной вместимости рюкзака проверяются рюкзаки меньшего размера (подзадачи). В данном случае, если дан рюкзак вместимостью 20 кг, алгоритм проверяет диапазон рюкзаков вместимостью от 0 до 20 кг.

2 Для каждого предмета проверяется, как он помещается в каждый из рюкзаков, начиная с самого маленького и заканчивая самым большим. При каждой проверке, если предмет помещается, выбирается лучшее значение из следующих вариантов:

а) решение, полученное для предыдущего меньшего рюкзака;

б) тестируемый предмет плюс заполнение оставшегося пространства лучшим из ранее найденных решений для этого объема.

Пример программы решает задачу о рюкзаке для следующего набора из шести предметов с разными комбинациями веса и стоимости, а также рюкзака вместимостью 20 кг:

Предмет	1	2	3	4	5	6
Вес в кг	2	3	4	4	5	9
Прибыль в 100 USD	3	4	3	5	8	10

Вот код для выполнения описанной процедуры динамического программирования (этот код можно найти в загружаемом файле исходного кода **A4D2E; 16; Knapsack.ipynb**; подробнее см. во введении):

```
values = [3, 4, 3, 5, 8, 10]
weights = [2, 3, 4, 4, 5, 9]
items = len(weights)
capacity = 20

memo = dict()
for size in range(0, capacity+1, 1):
    memo[(-1, size)] = ([], 0)

for item in range(items):
    for size in range(0, capacity+1, 1):
        # if the object doesn't fit in the knapsack
        if weights[item] > size:
            memo[item, size] = memo[item-1, size]
        else:
            # if the object fits, we check what can best fit
            # in the residual space
            previous_row, previous_row_value = memo[
                item-1, size-weights[item]]
            if memo[item-1, size][1] > values[item]
                + previous_row_value:
```



```

        memo[item, size] = memo[item-1, size]
    else:
        memo[item, size] = (previous_row + [item
                                     ], previous_row_value + values[item])

```

Лучшее решение — это кешированный результат, полученный при проверке вставки последнего предмета с полной вместимостью рюкзака (20 кг):

```

best_set, score = memo[items-1, capacity]
best_set_weights = [weights[item] for item in best_set]
print(f'The best set is {best_set}', end= ' ')
print(f'it weights {sum(best_set_weights)}', end= ' ')
print(f'and it values {score}')

```

The best set is [0, 3, 4, 5], it weights 20 and it values 26

Вам может быть интересно узнать, что происходило внутри словаря мемоизации:

```

print(len(memo))

147

print(memo[2, 10])

([0, 1, 2], 10)

knapsacks = len(range(0, capacity+1, 1))
print(f'tested {items} items, tested {knapsacks} knapsacks")

tested 6 items, tested 21 knapsacks

```

Он содержит 147 подзадач. Фактически шесть предметов, умноженные на 21 рюкзак (с вместимостью от 0 до 20), дают 126 решений, но нужно добавить еще 21 наивное решение, чтобы алгоритм работал правильно (*наивное* означает оставить рюкзак пустым), что увеличивает количество подзадач до 147.

Решение 147 подзадач может показаться пугающим (хотя они решаются молниеносно быстро). Использование только метода полного перебора для решения этой задачи означает решение меньшего количества подзадач в данном конкретном случае. Решение меньшего количества подзадач требует меньше времени, что можно проверить, решив задачу с помощью Python и функции `comb()`:

```

from scipy.special import comb

objects = 6
print(sum([comb(objects, k+1) for k in range(objects)]))

63.0

objects = 20
print(sum([comb(objects, k+1) for k in range(objects)]))

1048575.0

```

Для решения этой задачи требуется проверить 63 комбинации. Однако, если попробовать использовать больше объектов, скажем 20, время выполнения будет сильно различаться, поскольку теперь нужно проверить 1 048 575 комбинаций. Сравните это огромное число с динамическим программированием, которое требует решения всего $20 \times 21 + 21 = 441$ подзадачи (конечно, это значение верно только для рюкзаков, вмещающих 20 предметов).



ЗАПОМНИТЕ

Предыдущие результаты показывают разницу между квазиполиномиальным и экспоненциальным временем. (Напомним, что экспоненциальная сложность обсуждается в главе 2 при иллюстрации *Big-O*-нотации. В главе 15 вы узнаете о полиномиальном времени в рамках обсуждения NP-полных задач.) Использование динамического программирования становится плодотворным, когда задачи сложные. Учебные примеры хороши для обучения, но они не могут продемонстрировать весь потенциал применения умных алгоритмических методов, таких как динамическое программирование. Каждое решение проверяет, что происходит после добавления определенного предмета, когда рюкзак — определенного размера. В предыдущем примере добавляется предмет 2 (вес = 4, стоимость = 3) и выводится решение, которое помещает предметы 0, 1 и 2 в рюкзак (общий вес 9 кг) со стоимостью 10. Это промежуточное решение использует предыдущие решения и служит основой для многих последующих решений до завершения работы алгоритма.



СОВЕТ

Вы можете задаться вопросом, действительно ли результат, предложенный скриптом, является наилучшим из возможных. К сожалению, единственный способ убедиться в этом — знать правильный ответ, что означает необходимость запуска алгоритма полного перебора (когда это выполнимо с точки зрения времени работы на вашем компьютере). В этой главе мы не используем полный перебор для задачи о рюкзаке, но вы увидите подход с использованием полного перебора в рассматриваемой далее задаче коммивояжера.

Путешествие по городам

Задача коммивояжера (сокращенно — TSP) не менее известна, чем задача о рюкзаке. Она в основном используется в логистике и транспортировке, например для доставки товаров как одним транспортным средством, так и целым автопарком. В задаче коммивояжера требуется, чтобы торговый представитель посетил определенное количество городов и вернулся в исходный город (поскольку маршрут замкнутый, его называют туром), используя кратчайший возможный путь.

TSP похожа на задачи теории графов, но без явного указания ребер, поскольку все города взаимосвязаны. По этой причине для TSP обычно используется матрица расстояний в качестве входных данных — таблица, в которой города перечислены как в строках, так и в столбцах (квадратная матрица, поскольку количество строк совпадает с количеством столбцов). На пересечениях содержатся расстояния от города в строке до города в столбце. В различных вариантах TSP матрица может содержать время или расход топлива вместо расстояний.

TSP является NP-трудной задачей, но ее можно решить различными способами — как приближенными (эвристическими), так и точными (динамическое программирование). Проблема, как и с любой другой NP-трудной задачей, заключается во времени выполнения. Хотя вы можете рассчитывать на решения задачи, которые, как вы надеетесь, достаточно хороши (вы не можете быть уверены, кроме случаев с короткими маршрутами), но вы не можете знать наверняка, решая такие сложные задачи, как кругосветное путешествие: <http://www.math.uwaterloo.ca/tsp/world>. В следующем примере тестируются различные алгоритмы, такие как полный перебор, жадный алгоритм и динамическое программирование, на простом маршруте из шести городов, представленном в виде взвешенного графа (см. рисунок 16.1). (Вы можете найти этот код в загружаемом файле исходного кода **A4D2E; 16; TSP.ipynb**; подробнее см. во введении.)

```
import networkx as nx
import matplotlib.pyplot as plt
from random import seed

D = [[0,20,16,25,24],[20,0,12,12,27],
      [16,12,0,10,14],[25,12,10,0,20],
      [24,27,14,20,0]]

Graph = nx.Graph()
nodes = len(D[0])
Graph.add_nodes_from(range(nodes))
for i in range(nodes):
```



```

for j in range(nodes):
    Graph.add_edge(i, j, weight=D[j][i])

seed(2)
pos=nx.shell_layout(Graph)
draw_params = {'with_labels':True,
               'node_color':'skyblue',
               'node_size':800, 'width':2,
               'font_size':14}
nx.draw(Graph, pos, **draw_params)
labels = nx.get_edge_attributes(Graph, 'weight')
nx.draw_networkx_edge_labels(Graph,pos,
                             edge_labels=labels)
plt.show()

```

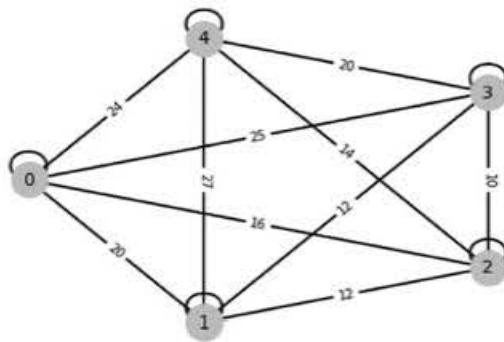


РИСУНОК 16.1
Города,
представлен-
ные как узлы
взвешенного
графа

После определения матрицы D (расстояний) в примере тестируется первое, простейшее решение для определения кратчайшего маршрута, начинающегося и заканчивающегося в городе с номером 0. Это решение основано на методе полного перебора, который генерирует все возможные перестановки порядка городов, исключая ноль, поскольку он является начальной и конечной точкой. Расстояние от 0 до первого города и от последнего города маршрута до 0 добавляется после подсчета общего расстояния для каждого решения. Когда все решения получены, просто выбирается кратчайший путь.

```

from itertools import permutations

best_solution = [None, sum([sum(row) for row in D])]
for solution in list(permutations(range(1, nodes))):
    start, distance = (0, 0)
    for next_one in solution:
        distance += D[next_one][start]
        start = next_one
    distance += D[0][start]
    if distance <= best_solution[1]:

```

```
best_solution = [[0]+list(solution)+[0], distance]
prt = str(best_solution)[1:-1]
print(f'Best solution so far: {prt} kms')
```

Best solution so far: [0, 1, 2, 3, 4, 0], 86 kms

Best solution so far: [0, 1, 3, 2, 4, 0], 80 kms

Best solution so far: [0, 4, 2, 3, 1, 0], 80 kms

Алгоритм полного перебора быстро определяет лучшее решение и его симметричный путь. Однако благодаря небольшому размеру задачи вы получаете быстрый ответ, поскольку для четырех городов существует всего 24 возможных решения. По мере увеличения количества городов число перестановок для проверки становится неприемлемо большим, даже после удаления симметричных путей (что сокращает число перестановок вдвое) и использования быстрого компьютера. Например, рассмотрим количество вычислений при работе с 13 городами плюс начальная или конечная точка:

```
from scipy.special import perm
```

```
print(perm(13, 13) / 2)
```

```
3113510400.0
```

Динамическое программирование может упростить время выполнения. Алгоритм Хелда — Карпа (также известный как алгоритм Беллмана — Хелда — Карпа, поскольку Беллман опубликовал его в 1962 году, в том же году, что и Майкл Хелд и Ричард Карп) может сократить временную сложность до $O(2n)$. Это по-прежнему экспоненциальная сложность, но требуется меньше времени, чем при полном переборе всех маршрутов методом грубой силы, у которого сложность — $O(n!)$.



СОВЕТ

Приближенные и эвристические алгоритмы могут обеспечить быстрые и полезные результаты. (Хотя результат не всегда может отражать оптимальное решение, обычно он достаточно хорош.) Вы снова встретитесь с задачей коммивояжера позже (см. главы 18 и 20), когда будем рассматривать локальный поиск и эвристики.

Чтобы найти лучшее решение задачи коммивояжера для n городов, начиная и заканчивая в городе с номером 0, алгоритм движется от города 0 и ведет учет кратчайшего возможного пути, рассматривая различные варианты. Он всегда использует разные конечные города и затрагивает только подмножество городов. По мере увеличения подмножеств алгоритм учится эффективно решать задачу. Таким образом, при решении задачи коммивояжера для пяти городов алгоритм сначала рассматривает

решения с двумя городами, затем с тремя городами, затем с четырьмя и, наконец, с пятью. (Множества имеют размерности от 1 до n .) Вот шаги, которые использует алгоритм:

- 1 Инициализировать таблицу для отслеживания расстояний от города 0 до всех остальных городов. Эти множества содержат только начальный город и город назначения, поскольку они представляют начальный шаг.
- 2 Рассмотреть все возможные размеры множеств, от двух до количества городов в маршруте. Это первая итерация, внешний цикл.
- 3 Внутри внешнего цикла для каждого размера множества рассмотреть все возможные комбинации городов этого размера, не содержащие начальный город. Это внутренняя итерация.
- 4 Внутри внутренней итерации (шаг 3) для каждой доступной комбинации рассмотреть каждый город внутри комбинации как конечный город. Это еще одна внутренняя итерация.
- 5 Внутри внутренней итерации (шаг 4) для каждого конечного города определить кратчайший путь, соединяющий города во множестве от начального города (город 0). При поиске кратчайшего пути используется любая полезная, ранее сохраненная информация, таким образом применяя динамическое программирование. Этот шаг экономит вычисления и обосновывает работу с растущими подмножествами городов. Повторно используя ранее решенные подзадачи, вы находите более короткие маршруты, добавляя к предыдущему кратчайшему пути расстояние, необходимое для достижения конечного города. Для определения набора городов, конкретного начального города и конкретного конечного города алгоритм сохраняет лучший путь и его длину.
- 6 Когда все итерации заканчиваются, есть столько различных кратчайших решений, сколько городов $n - 1$, причем каждое решение охватывает все города, но заканчивается в разных городах. Добавьте конечную точку, город 0, к каждому из них, чтобы завершить маршрут.
- 7 Определить кратчайшее решение и вывести его как результат.



ПРИМЕЧАНИЕ

Реализация этого алгоритма на Python не очень проста, поскольку включает несколько итераций и манипуляции со множествами. Это полный перебор, усиленный динамическим программированием, который опирается на итеративный подход с подмножествами городов и кандидатами для добавления к ним. Следующий прокомментированный пример на Python показывает, как работает это решение. Вы можете использовать его для расчета пользовательских маршрутов (возможно, используя города вашего региона или округа как элементы в матрице расстояний). Скрипт использует продвинутые команды, такие как `frozenset()` (коман-

да, которая делает множество пригодным для использования в качестве ключа словаря) и операторы для множеств для достижения решения:

```
from itertools import combinations

memo = {(frozenset([0, idx+1]), idx+1): (dist, [0,idx+1])
        for idx,dist in enumerate(D[0][1:])}

cities = nodes
for subset_size in range(2, cities):
    # Here we define the size of the subset of cities
    new_memo = dict()
    for subset in [frozenset(comb) | {0} for comb in
                   combinations(range(1, cities),
                               subset_size)]:
        # We enumerate the subsets having a certain subset
        # size
        for ending in subset - {0}:
            # We consider every ending point in the subset
            all_paths = list()
            for k in subset:
                # We check the shortest path for every
                # element in the subset
                if k != 0 and k!=ending:
                    length = memo[(subset-{ending},k)][0]
                               ] + D[k][ending]
                    index  = memo[(subset-{ending},k)][1]
                               ] + [ending]
                    all_paths.append((length, index))
            new_memo[(subset, ending)] = min(all_paths)
        # In order to save memory, we just record the previous
        # subsets since we won't use shorter ones anymore
        memo = new_memo
    # Now we close the cycle and get back to the tour start
    # of the tour, city zero
    tours = list()
    for distance, path in memo.values():
        distance += D[0][path[-1]]
        tours.append((distance, path + [0]))
    # We can now declare the shortest tour
    distance, path = min(tours)
    print('Shortest dynamic programming tour is:', end=' ')
    print(f'{path}, {distance} kms')

Shortest dynamic programming tour is:
[0, 1, 3, 2, 4, 0], 80 kms
```

Приближенный поиск строк

Определить, когда одно слово похоже на другое, не всегда просто. Слова могут немного различаться из-за орфографических ошибок или разных способов написания самого слова, что делает точное совпадение невозможным. Это не просто проблема, которая поднимает интересные вопросы при проверке орфографии. Например, объединение похожих текстовых строк (таких, как имена, адреса или идентификаторы кода), относящихся к одному и тому же человеку, может помочь создать единое представление о клиентской базе компании или помочь службе национальной безопасности найти опасного преступника.



ЗАПОМНИТЕ

Приближенный поиск строк находит широкое применение в машинном переводе, распознавании речи, проверке орфографии, обработке текста, вычислительной биологии и информационном поиске. Анализируя способы ввода данных в базы данных, вы понимаете, что между полями данных есть много несоответствий, которые должен решать умный алгоритм. Сопоставление похожих, но не абсолютно идентичных последовательностей букв — это возможность, которая находит применение в таких областях, как генетика, где сравниваются последовательности ДНК (выраженные буквами, представляющими нуклеотиды *G*, *A*, *T* и *C*), чтобы определить, насколько две последовательности похожи и как они соотносятся друг с другом.

Владимир Левенштейн, российский ученый, эксперт в области теории информации (подробнее см. на http://ethw.org/Vladimir_I._Levenshtein), в 1965 году разработал простую меру (названную его именем), которая вычисляет степень сходства между двумя строками, подсчитывая, сколько преобразований требуется для превращения первой строки во вторую. *Расстояние Левенштейна* (также известное как *редакционное расстояние*) подсчитывает, сколько изменений нужно сделать в слове:

- » **удаление** – удаление буквы из слова;
- » **вставка** – вставка буквы в слово с получением другого слова;
- » **замена** – замена одной буквы на другую, например замена буквы *p* на букву *f* для получения *fan* из *rap*.



СОВЕТ

У каждой операции редактирования своя стоимость, которую Левенштейн определяет как 1 для каждого преобразования. Однако в зависимости от того, как вы применяете алгоритм, вы можете по-разному задавать стоимость удаления, вставки и замены. Например, при поиске похожих названий улиц опечатки встречаются чаще, чем явные различия в написании, поэтому замена может стоить всего 1, а удаление или вставка — 2. С другой стороны, при поиске денежных сумм у похожих

значений вполне может быть разное количество цифр. Кто-то может ввести в базу данных 123 \$ или 123,00 \$. Числа одинаковые, но количество цифр разное, поэтому вставка и удаление могут стоить меньше, чем замена. (Значение 124 \$ не совсем то же самое, что значение 123 \$, поэтому замена 3 на 4 должна стоить дороже.)

Алгоритм подсчета можно реализовать как рекурсивно, так и итеративно. Однако он работает гораздо быстрее при использовании восходящего решения с помощью динамического программирования, как описано в статье 1974 года «The String-to-string Correction Problem» Роберта А. Вагнера и Майкла Дж. Фишера (<http://www.inrg.csie.ntu.edu.tw/algorithm2014/homework/Wagner-74.pdf>). Временная сложность этого решения составляет $O(mn)$, где n и m — длины сравниваемых слов в буквах. Следующий код вычисляет количество изменений, необходимых для преобразования слова *Saturday* в *Sunday*, используя динамическое программирование с матрицей (см. рисунок 16.2) для хранения предыдущих результатов (подход снизу вверх). (Этот код можно найти в загружаемом файле исходного кода A4D2E; 16; Levenshtein.ipynb; подробнее см. во введении.)

```
s1 = 'Saturday'
s2 = 'Sunday'
m = len(s1)
n = len(s2)
D = [list(range(n+1))]
for i in range(m):
    D.append([i+1] + [0] * n)

for j in range(1, n+1):
    for i in range(1, m+1):
        if s1[i-1] == s2[j-1]:
            D[i][j] = D[i-1][j-1]
        else:
            deletion = D[i-1][j] + 1
            insertion = D[i][j-1] + 1
            substitution = D[i-1][j-1] + 1
            D[i][j] = min(deletion,
                          insertion,
                          substitution)
print (f'Levenshtein distance is {D[-1][-1]}')

Levenshtein distance is 3
```


Вы можете построить граф или вывести результат с помощью Pandas (пакет для анализа данных и визуализации). Pandas напечатает аккуратный заголовок и метки строк для матрицы, представленной списком списков **D**:

```
import pandas as pd
pd.DataFrame(D, columns=list(' '+s2), index=list(' '+s1))
```

		S	u	n	d	a	y
	0.0	1.0	2.0	3.0	4.0	5.0	6.0
S	1.0	0.0	1.0	2.0	3.0	4.0	5.0
a	2.0	1.0	1.0	2.0	3.0	3.0	4.0
t	3.0	2.0	2.0	2.0	3.0	4.0	4.0
u	4.0	3.0	2.0	3.0	3.0	4.0	5.0
r	5.0	4.0	3.0	3.0	4.0	4.0	5.0
d	6.0	5.0	4.0	4.0	3.0	4.0	5.0
a	7.0	6.0	5.0	5.0	4.0	3.0	4.0
y	8.0	7.0	6.0	6.0	5.0	4.0	3.0

РИСУНОК 16.2.
Преобразова-
ние Saturday
в Sunday

Возвращаясь к алгоритму: он строит матрицу, помещая оптимальное решение в последнюю ячейку. После построения матрицы, где буквы первой строки используются как строки, а буквы второй — как столбцы, алгоритм продвигается по столбцам, вычисляя различия между каждой буквой в строках по сравнению с буквами в столбцах. Таким образом, алгоритм выполняет количество сравнений, равное произведению количества букв в двух строках. По мере продвижения алгоритм учитывает результаты предыдущих сравнений и выбирает решение с наименьшим количеством правок.

Когда итерация по матрице завершается, полученное число представляет минимальное количество правок, необходимых для осуществления преобразования, — чем меньше число, тем более похожи две строки. Обратное прослеживание от последней ячейки к первой путем перехода к предыдущей ячейке с наименьшим значением (если доступно

несколько направлений, то предпочтительнее двигаться по диагонали) подсказывает, какие преобразования нужно выполнить (см. рисунок 16.3). Обратите внимание, что путь необязательно уникален. Подчеркивания показывают, где доступны два возможных варианта.

- » Диагональное движение назад указывает на замену в первой строке, если буквы в строке и столбце различаются (в противном случае правка не требуется).
- » Движение вверх указывает на удаление буквы в первой строке.
- » Движение влево и назад означает, что в первую строку нужно вставить новую букву.

		S	u	n	d	a	y
	0.0	1.0	2.0	3.0	4.0	5.0	6.0
S	1.0	<u>0.0</u>	1.0	2.0	3.0	4.0	5.0
a	2.0	<u>1.0</u>	1.0	2.0	3.0	3.0	4.0
t	3.0	<u>2.0</u>	2.0	2.0	3.0	4.0	4.0
u	4.0	3.0	2.0	3.0	3.0	4.0	5.0
r	5.0	4.0	3.0	<u>3.0</u>	4.0	4.0	5.0
d	6.0	5.0	4.0	4.0	3.0	4.0	5.0
a	7.0	6.0	5.0	5.0	4.0	3.0	4.0
y	8.0	7.0	6.0	6.0	5.0	4.0	3.0

РИСУНОК 16.3.
Выделение
применяемых
преобразо-
ваний

В этом примере обратное отслеживание указывает на следующие преобразования (два удаления и одна замена):

Saturday => Sturday => Surday => Sunday

В ЭТОЙ ГЛАВЕ

- » Понимание того, как случайность может оказаться умнее продуманных способов решения
- » Введение в ключевые идеи вероятности и ее распределений
- » Ознакомление с работой метода Монте-Карло
- » Изучение алгоритма быстрого выбора и возвращение к алгоритму быстрой сортировки

ГЛАВА 17

Использование рандомизированных алгоритмов

Генераторы случайных чисел — ключевая функция в вычислении, они играют важную роль в алгоритмических методах, рассматриваемых в этой части книги. Как описано в первом параграфе главы, рандомизация используется не только для игр или азартных развлечений — она применяется и для решения широкого спектра задач. Иногда рандомизация оказывается при оптимизации эффективнее других методов и позволяет получить правильное решение быстрее, чем более рациональные подходы. Она помогает улучшить работу различных техник, таких как локальный поиск, имитация отжига для эвристик, криптография и распределенные вычисления (при этом криптография для сокрытия информации наиболее критична).

В блоке «Понимание работы вероятности» рассматриваются базовые принципы теории вероятностей, а затем объясняется, как распределение вероятностей связано с рандомизацией. На примерах карточной игры и алгоритма быстрого выбора во втором параграфе главы демонстриру-

ется, как рандомизация может находить более короткие пути и достигать результата проще, чем другие алгоритмы.

Фактически рандомизация упрощает поиск решения, обменивая время на сложность. Упрощение задач — не единственное ее преимущество: рандомизация экономит ресурсы и работает распределенно с меньшей потребностью в коммуникации и координации. Эта глава знакомит вас с информацией, необходимой для понимания того, как обогащение алгоритмов случайностью может помочь в решении задач (в главе используется термин *внедрение случайности*, как если бы это было лекарство). Еще больше применений ждет в следующих главах, поэтому здесь рассматриваются и такие ключевые темы, как основы теории вероятностей, распределение вероятностей и моделирование методом Монте-Карло.



ЗАПОМНИТЕ

Вам не нужно вводить исходный код для этой главы вручную. На самом деле, использовать загружаемый исходный код намного проще. Исходный код главы можно найти в файлах **A4D2E; 17; Probability.ipynb** и **A4D2E; 17; Quickselect.ipynb** из загружаемых материалов. Подробнее о том, как найти эти исходные файлы, смотрите во введении.

§ 1. Определение принципа работы рандомизации

Элементы случайности можно неожиданно обнаружить во многих повседневных устройствах. Первые версии робота-пылесоса Roomba (разработанного компанией, основанной выпускниками Массачусетского технологического института) убирали помещения без четкого плана или схемы пространства. Устройство в основном работало, случайным образом перемещаясь по комнате, и, согласно оригинальному патенту, после столкновения с препятствием поворачивалось на случайное количество градусов и начинало двигаться в новом направлении. Тем не менее Roomba всегда успешно справлялся с уборкой. (Если вам интересно узнать, как он работал и как за эти годы усовершенствовал свои навигационные возможности, посетите страницу <https://www.explainthatstuff.com/how-roomba-works.html>.)

С исторической точки зрения рандомизированные алгоритмы — это относительно недавнее изобретение: первый алгоритм такого типа, алгоритм поиска ближайшей пары точек, был разработан Майклом Рабином в 1976 году. Этот алгоритм определяет среди множества точек на геометрической плоскости пару точек, между которыми наименьшее расстояние, причем делает это без необходимости сравнивать

все точки между собой. За алгоритмом ближайшей пары в следующем году последовал рандомизированный тест простоты числа (алгоритм для определения, является ли число составным или вероятно простым), разработанный Робертом М. Соловеем и Фолькером Штрассеном. Вскоре после этого применение в криптографии и распределенных вычислениях сделало рандомизацию более популярной и предметом интенсивных исследований, хотя эта область все еще остается новой и во многом неизученной.

Рандомизация опирается на способность компьютера генерировать случайные числа, то есть создавать числа без определенного плана. Следовательно, случайное число непредсказуемо, и при генерации последующих случайных чисел они не должны быть связаны друг с другом.



ЗАПОМНИТЕ

Однако достичь истинной случайности сложно. Даже когда вы бросаете кости, результат не может быть полностью неожиданным из-за того, как вы держите кости, как вы их бросаете и того факта, что кости неидеальной формы. Компьютеры тоже не очень хороши в создании случайных чисел. Они генерируют случайность, используя алгоритмы или таблицы псевдослучайных чисел (которые работают, используя начальное значение, или *зерно*, в качестве отправной точки, что эквивалентно индексу), поскольку компьютер не может создать по-настоящему случайное число. Компьютеры — это детерминированные машины; все внутри них следует четко определенному шаблону реакции, а это означает, что они лишь имитируют случайность определенным образом.

Рассмотрим, почему нужна рандомизация

Даже если компьютер не может создать истинную случайность, потоки *псевдослучайных чисел* (чисел, которые кажутся случайными, но каким-то образом предопределены) все равно могут иметь решающее значение во многих задачах информатики. Любой алгоритм, использующий случайность в своей логике, может считаться рандомизированным, независимо от того, определяет ли случайность его результаты, улучшает производительность или снижает риск неудачи, предоставляя решение в определенных случаях.

Обычно случайность используется при выборе входных данных, начальной точки оптимизации или количества и типа операций, применяемых к данным. Когда случайность — ключевая часть логики алгоритма, а не просто средство улучшения его производительности, ожидаемое время выполнения алгоритма и даже его результаты могут стать неопределенными и подверженными случайности. Например, алгоритм может выдавать разные, хотя и одинаково хорошие, результаты при

каждом запуске. Поэтому полезно различать типы рандомизированных решений, каждое из которых названо в честь знаковых мест для азартных игр.

- » **Лас-Вегас.** Эти алгоритмы примечательны тем, что используют случайные входные данные или ресурсы для получения правильного ответа на задачу каждый раз. Получение результата может занять неопределенное время из-за случайных процедур. Примером служит алгоритм быстрой сортировки.
- » **Монте-Карло.** Из-за использования случайности алгоритмы Монте-Карло могут не предоставить правильного ответа или вообще не дать ответа, хотя такие исходы случаются редко. Поскольку результат неопределен, их время выполнения может быть ограничено максимальным числом испытаний. Алгоритмы Монте-Карло демонстрируют, что алгоритмы необязательно всегда успешно решают задачи, для которых они предназначены. Примером служит тест простоты числа Соловея – Штрассена.
- » **Атлантик-Сити.** Эти алгоритмы работают за полиномиальное время, предоставляя правильный ответ на задачу как минимум в 75% случаев. Алгоритмы Монте-Карло всегда быстрые, но не всегда правильные, а алгоритмы Лас-Вегаса всегда правильные, но не всегда быстрые. Поэтому алгоритмы Атлантик-Сити считаются промежуточными между ними, так как они обычно и быстрые, и правильные.

Этот класс алгоритмов был представлен в 1982 году Дж. Финном в неопубликованной рукописи под названием «Сравнение вероятностных тестов на простоту чисел». Созданный по теоретическим причинам для проверки простых чисел, класс включает сложные в разработке решения, поэтому сегодня их очень мало.

Понимание работы вероятности

Вероятность показывает шанс события, который обычно выражается числом. В этой книге, как и в целом в области вероятностных исследований, вероятность события измеряется в диапазоне от 0 (нулевая вероятность наступления события) до 1 (полная уверенность в наступлении события). Промежуточные значения, такие как 0,25 или 0,75, указывают, что событие будет происходить с определенной частотой при условиях, которые могут привести к этому событию (называемых *испытаниями*). Даже если

числовой диапазон от 0 до 1 поначалу не кажется интуитивно понятным, работа с вероятностью со временем делает причину использования такого диапазона понятнее. Когда событие происходит с вероятностью 0,25, вы знаете, что из 100 испытаний событие, скорее всего, произойдет около $0,25 \times 100 = 25$ раз.

Например, когда вероятность победы вашей любимой спортивной команды составляет 0,75, вы можете использовать это число, чтобы определить шансы на успех, когда ваша команда играет против другой команды. Вы даже можете получить более конкретную информацию, например вероятность победы в определенном турнире (у вашей команды вероятность выиграть матч в этом турнире составляет 0,65) или при определенных условиях (когда команда играет на выезде, вероятность победы снижается до 0,60).

Вероятности могут многое рассказать о событии, и они также полезны для алгоритмов. При использовании рандомизированного алгоритмического подхода может возникнуть вопрос, когда остановить алгоритм, поскольку он должен был найти решение. Важно знать, как долго искать решение, прежде чем сдаться. Вероятности могут помочь определить, сколько итераций может потребоваться. В главе 18 обсуждение алгоритма *2-выполнимости* (или 2-SAT) предоставляет рабочий пример использования вероятностей в качестве правил остановки алгоритма.



ЗАПОМНИТЕ

О вероятностях часто говорят в процентах в спорте и экономике, указывая, что событие происходит определенное количество раз после 100 испытаний. Это в точности та же вероятность, независимо от того, выражаете вы ее как 0,25 или как 25%. Это просто вопрос условностей. В азартных играх вы даже слышите о шансах — это другой способ выражения вероятности, где сравнивается вероятность наступления события (например, победы определенной лошади в скачках) с вероятностью того, что событие не произойдет вообще. В этом случае 0,25 выражается как 25 против 75, 25 к 75 или любым другим способом, приводящим к тому же соотношению.

Вы можете умножить вероятность на количество испытаний и получить ожидаемое число появлений события, но, действуя наоборот, можно эмпирически оценить вероятность. Проведите определенное количество испытаний, наблюдайте за каждым из них и подсчитайте, сколько раз происходит событие. Отношение между количеством появлений и количеством испытаний — это ваша оценка вероятности. Например, вероятность 0,25 — это вероятность выбрать определенную масть при случайном выборе карты из колоды. Французские игральные карты (наиболее широко используемая колода; она также распространена в Америке и Британии) представляют собой классический пример для объяснения вероятностей. (Итальянцы, немцы и швейцарцы, например, исполь-

зуют колоды с другими мастями, о которых вы можете прочитать на <http://healthy.uwaterloo.ca/museum/VirtualExhibits/Playing%20Cards/decks/index.html>.) Колода содержит 52 карты, равномерно распределенные по четырем мастям: трефам и пикам, которые черные, и бубнам и червам, которые красные. Если хотите определить вероятность вытянуть туза, нужно учесть, что в колоде четыре туза. Количество возможных испытаний при вытягивании карт равно 52 (количество карт), поэтому ответ в терминах вероятности — $4/52 = 0,077$.



СОВЕТ

Более надежную оценку эмпирической вероятности можно получить, увеличив количество испытаний. При небольшом числе испытаний оценка вероятности события может быть неточной из-за влияния случайности. По мере увеличения количества испытаний наблюдаемые значения будут приближаться к истинной вероятности самого события. В основе этого лежит принцип, что за событиями стоит некий порождающий процесс. Чтобы понять, как работает этот порождающий процесс, требуется множество испытаний. Такое использование испытаний также известно как *выборка* из вероятностного распределения.

Понимание распределений

Распределение вероятностей — это еще одна важная концепция для разработки более эффективных алгоритмов. *Распределение* — это таблица значений или математическая функция, которая связывает каждое возможное значение входных данных с вероятностью появления таких значений. Распределения вероятностей обычно (но не всегда) представляются в виде графиков, где ось абсцисс отображает возможные значения входных данных, а ось ординат — вероятность их появления. Большинство статистических моделей основаны на нормальном распределении — симметричном распределении, имеющем характерную колоколообразную форму. Для представления нормального распределения в Python (как показано на рисунке 17.1) требуется несколько строк кода. (Этот код можно найти в загружаемом файле исходного кода **A4D2E; 17; Probability.ipynb**; подробнее см. во введении.) Обратите внимание, как код заставляет компьютер генерировать одни и те же случайные числа, используя команду `seed(0)`, которая вызывает определенную последовательность случайных чисел, хранящуюся в памяти компьютера. Как упоминалось ранее, компьютеры детерминированы и случайность может быть только имитирована определенным образом.


```

from random import seed, gauss, uniform
import matplotlib.pyplot as plt

seed(0)
normal_distribution = [gauss(mu=25, sigma=100)
                       for r in range(10_000)]
weights = [1./10_000] * 10_000

plt.figure(figsize=(12, 6))
plt.hist(normal_distribution, bins=20, weights=weights,
         color='lightblue', edgecolor='black',
         linewidth=1.2)
plt.xlabel("Value")
plt.ylabel("Probability")
plt.show()

```

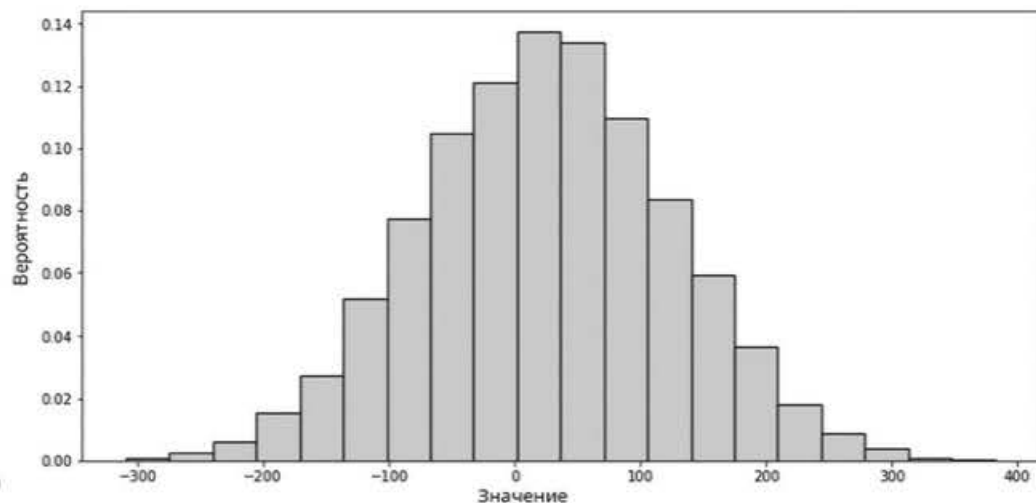


РИСУНОК 17.1.
Гистограмма
нормального
распределения

Построенное распределение представляет входные данные из 10 000 чисел, среднее значение которых — около 100. Каждый столбец гистограммы представляет вероятность появления определенного диапазона значений во входных данных. Если сложить все столбцы, получится значение 1, которое включает все вероятности, выраженные распределением.

При нормальном распределении большинство значений группируется вокруг среднего значения. Поэтому, если вы случайным образом выбираете число из входных данных, с наибольшей вероятностью вы получите число около центра распределения. Числа, далекие от центра, будут выпадать реже. Если ваш алгоритм при использовании среднего значения работает лучше, чем с любым другим числом, то выбор случайного числа из распределения с центром в среднем значении имеет смысл и может быть проще, чем разработка более сложного способа обработки входных данных.

Другое важное распределение, упомянутое в этой главе, — равномерное распределение. Его также можно представить с помощью кода на Python (результат показан на рисунке 17.2):

```
seed(0)
uniform_distribution = [uniform(a=0, b=100)
                        for r in range(10_000)]
weights = [1./10_000] * 10_000

plt.figure(figsize=(12, 6))
plt.hist(uniform_distribution, bins=20, weights=weights,
         color='lightblue', edgecolor='black',
         linewidth=1.2)
plt.xlabel("Value")
plt.ylabel("Probability")
plt.show()
```

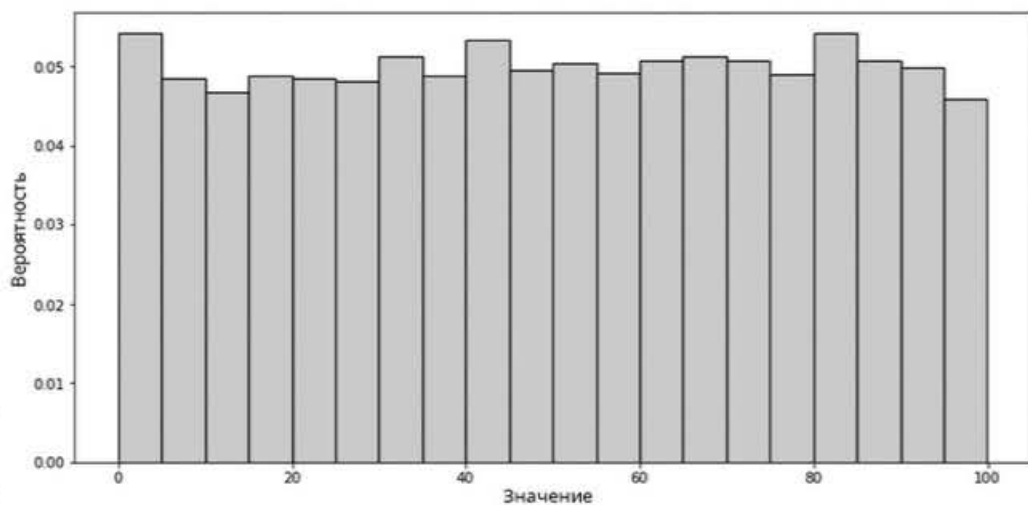


РИСУНОК 17.2.
Гистограмма
равномерного
распределения

Равномерное распределение заметно отличается от нормального тем, что у каждого числа есть одинаковая вероятность появления во входных данных. Как следствие, столбцы гистограммы примерно одинаковой высоты, а выбор числа из равномерного распределения означает, что у всех чисел равные шансы быть выбранными. Это способ избежать систематического выбора одних и тех же групп чисел, когда ваш алгоритм лучше работает с разнообразными входными данными. Например, равномерное распределение хорошо работает, когда ваш алгоритм отлично справляется с определенными числами, средне — когда с большинством, и плохо — когда с некоторыми другими, и вы предпочитаете выбирать числа случайным образом, чтобы избежать выбора серии «плохих» чисел. Эта стратегия используется в алгоритмах быстрого выбора и рандомизированной быстрой сортировки, описанных далее в главе.

Поскольку алгоритмам нужны числовые входные данные, знание их распределения может помочь сделать их работу эффективнее. Важно не только начальное распределение. Вы также можете использовать то, как распределение данных меняется по мере работы алгоритма.

В качестве примера, как изменяющееся распределение может улучшить ваш алгоритм, следующий код показывает, как угадать карту во французской колоде с помощью случайного выбора:

```
numbers = ['Ace', '2', '3', '4', '5', '6', '7', '8', '9',
           '10', 'Jack', 'Queen', 'King']
seeds = ['Clubs', 'Spades', 'Diamonds', 'Hearts']
deck = [s+'_'+n for n in numbers for s in seeds]

from random import choice

seed(0)
my_cards = deck.copy()
guessed = 0
for card in deck:
    if card == choice(my_cards):
        guessed += 1
print('Guessed %i card(s)' % guessed)

Guessed 2 card(s)
```

Код начинается с создания колоды (*deck*) путем комбинации возможных значений карт (от туза до короля) и мастей (от треф до червей) с помощью списковых включений Python, так что результирующая колода содержит значения вроде *'Hearts_King'* и *'Clubs_Ace'*. Затем код импортирует функцию *choice()*, которая может случайным образом выбрать элемент из колоды. Вызов *seed(0)* гарантирует получение одинакового результата при каждом запуске, но обычно не стоит устанавливать начальное значение так, чтобы результат действительно выглядел случайным. Следующий шаг — создание копии колоды, *my_cards*, для сравнения. Затем код использует цикл для последовательного перебора каждой карты в колоде и получает случайную карту из *my_cards* с помощью вызова *choice()*. Когда карты совпадают, значение *guessed* увеличивается.

Такая стратегия приносит скромные результаты — в среднем вы угадаете всего одну карту за все 52 попытки. Действительно, в каждой попытке вероятность угадать правильную карту составляет $1/52$, что в сумме дает 1 после перебора всех карт: $(1/52) \times 52 = 1$. Вместо этого можно изменить этот простой случайный алгоритм, исключая из возможных вариантов уже просмотренные карты:

```
seed(0)
my_cards = deck.copy()
guessed = 0
for card in deck:
    if card == choice(my_cards):
        guessed += 1
    else:
        my_cards.pop(my_cards.index(card))
print('Guessed %i card(s)' % guessed)

Guessed 3 card(s)
```

Теперь в среднем вы будете угадывать правильную карту чаще, потому что по мере уменьшения колоды ваши шансы на правильный выбор увеличиваются и вероятность угадывания существенно возрастает ближе к концу игры. (Ваши шансы равны единице, деленной на количество оставшихся в колоде карт.)



СОВЕТ

Подсчет карт может дать преимущество в карточных играх. Группа студентов MIT использовала подсчет карт и вероятностные оценки, чтобы выиграть огромные суммы в Лас-Вегасе, пока эта практика не была запрещена в казино. Эта история даже вдохновила создателей на съемку фильма *«Двадцать одно»* (2008) с Кевином Спейси в главной роли. Подробнее об этой истории можно прочитать на <https://www.bbc.com/news/magazine-27519748>.

Моделирование использования метода Монте-Карло

Расчет вероятностей, помимо операций, рассмотренных в этой главе, выходит за рамки книги. Понять, как работает алгоритм с элементами случайности, непросто даже при знании того, как вычислять вероятности, поскольку он может быть результатом смещения множества различных распределений вероятностей. Однако обсуждение метода Монте-Карло проливает свет на результаты самых сложных алгоритмов и помогает понять принципы их работы. Этот метод используется как в математике, так и в физике для решения многих задач. Например, такие ученые, как Энрико Ферми и Эдвард Теллер, использовали моделирование по методу Монте-Карло на специально разработанных суперкомпьютерах во время Манхэттенского проекта (в рамках которого во время Второй мировой войны разрабатывалась атомная бомба) для ускорения своих экспериментов. Подробнее об этом можно прочитать на <https://www.atomicheritage.org/history/computing-and-manhattan-project>.



Не следует путать метод Монте-Карло с алгоритмом Монте-Карло. Метод Монте-Карло — это способ понять, как распределение вероятностей влияет на задачу, тогда как алгоритм Монте-Карло, как обсуждалось ранее, — это рандомизированный алгоритм, который не гарантирует нахождения решения.

В симуляции Монте-Карло вы многократно берете выборки результатов работы алгоритма. Вы сохраняете определенное количество результатов, затем вычисляете статистические показатели, такие как среднее значение, и визуализируете их в виде распределения. Например, если хотите лучше понять, как уменьшение размера колоды, из которой вы берете карты, может помочь достичь лучших результатов (как в предыдущем скрипте Python), вы несколько раз запускаете алгоритм и записываете частоту успешных исходов:

```
import numpy as np
seed(0)
samples = list()
for trial in range(10_000):
    my_cards = deck.copy()
    guessed = 0
    for card in deck:
        if card == choice(my_cards):
            guessed += 1
        else:
            my_cards.pop(my_cards.index(card))
    samples.append(guessed)
```

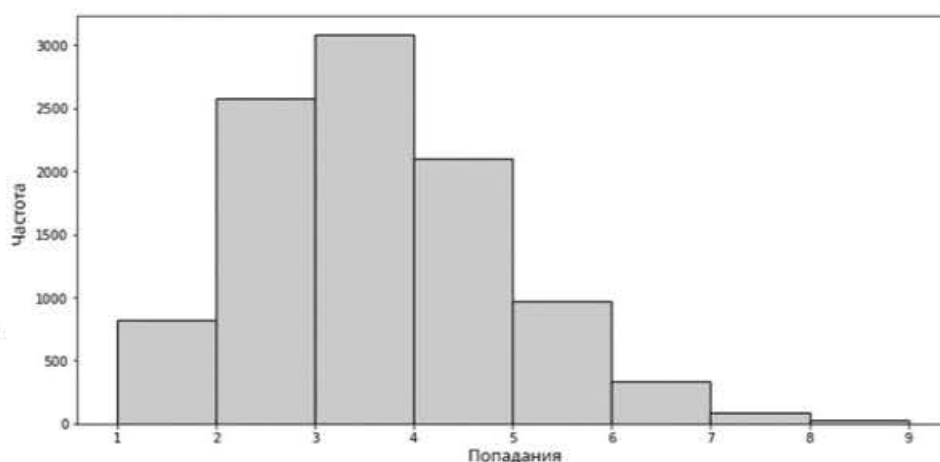
Выполнение симуляции Монте-Карло может занять несколько секунд. Необходимое время зависит от скорости работы алгоритма, размера задачи и количества испытаний. Однако при выборке из распределений чем больше испытаний вы проводите, тем стабильнее становится результат. В этом примере выполняется 10 000 испытаний. Вы можете как оценить, так и визуализировать ожидаемый результат (см. рисунок 17.3), используя следующий код:

```
plt.figure(figsize=(12, 6))
plt.hist(samples, bins=8, color='lightblue',
         edgecolor='black', linewidth=1.2)
plt.xlabel("Guesses")
plt.ylabel("Frequency")
plt.show()
mean = round(sum(samples) / len(samples), 2)
print(f'On average you can expect {mean} guesses each run')

On average you can expect 3.13 guesses each run
```

Анализируя полученную гистограмму, можно определить, что результат «три» получается примерно в 3000 запусках из 10000 испытаний, что делает его наиболее вероятным исходом. Интересно, что результат «ноль» не встречается ни разу, но также редко удастся набрать семь или более попаданий. В последующих примерах главы симуляции Монте-Карло используются для понимания работы более сложных рандомизированных алгоритмов.

РИСУНОК 17.3.
Отображение
результатов
симуляции
Монте-Карло



§ 2. Внесение случайности в вашу логику

Вот несколько из многих причин включать элемент случайности в логику вашего алгоритма:

- » это улучшает работу алгоритмов и позволяет находить более разумные решения;
- » требуется меньше ресурсов с точки зрения памяти и вычислений;
- » создаются алгоритмы, которые дают распределенный результат с минимальным контролем или вообще без него.

В следующей главе, посвященной локальному поиску, вы увидите, как рандомизация и вероятность могут оказаться полезными в ситуациях, когда сложно определить, в каком направлении должен двигаться ваш алгоритм. Примеры в последующих блоках демонстрируют, как рандомизация помогает быстро находить значения в определенной позиции во входных данных, а также как использование случайности может ускорить сортировку.

Вычисление медианы с помощью быстрого выбора

Вычисление статистической меры — медианы — может оказаться сложной задачей при работе с неотсортированными списками входных данных. Действительно, медиана зависит от позиции данных в упорядоченном виде:

- » если входные данные содержат нечетное количество элементов, медианой является точно среднее значение;
- » если входные данные содержат четное количество элементов, медиана представляет собой среднее значение пары средних чисел в упорядоченном списке.



ЗАПОМНИТЕ

Медиана, как и среднее значение, представляет собой единственное число, характеризующее распределение значений. В отличие от среднего, медиана, основанная на порядке элементов во входном векторе, практически не зависит от конкретных значений в списке — это просто среднее значение. При этом значения в начале и конце входного списка могут существенно повлиять на среднее арифметическое, если они крайне малы или велики. Такая устойчивость делает медиану очень полезной во многих статистических задачах. Простой пример вычисления медианы с помощью функций Python поможет понять суть этой меры. (Код можно найти в файле **A4D2E; 17; Quickselect.ipynb** из загружаемых исходников; подробнее см. во введении.)

```
from random import randint, random
from random import choice, seed
import sys
sys.setrecursionlimit(1500)

n = 501
seed(0)
series = [randint(1,25) for i in range(n)]

from statistics import median
# https://docs.python.org/3/library/statistics.html
print(f'Median is {median(series)}')

Median is 12.0
```

Код создает список из 501 элемента и находит медиану списка с помощью функции `median()` из пакета NumPy. Полученная медиана фактически является средней точкой упорядоченного списка, то есть 251-м элементом.


```
elem_251 = sorted(series)[250]
print(f'251st element of the ordered series is {elem_251}')
```

251st element of the ordered series is 12

Упорядочивание списка и извлечение нужного элемента показывает, как работает функция `median()`. Поскольку для вычисления медианы требуется сортировка, лучшее время выполнения составляет $O(n \cdot \log(n))$. Используя рандомизацию, предоставляемую алгоритмом быстрого выбора, можно получить даже лучший результат — время выполнения $O(n)$. Алгоритм быстрого выбора работает рекурсивно, поэтому в Python нужно установить более высокий предел рекурсии с учетом списка и позиции искомого значения в отсортированном списке. Индекс значения обозначается как k , и алгоритм также известен как алгоритм поиска k -го по величине элемента. Для получения результата он использует следующие шаги:

- 1 Выбирает опорный элемент в списке данных и разделяет список на две части: левую, содержащую числа меньше опорного, и правую — с числами больше опорного.
- 2 Определяет длину каждого списка. Если длина левого списка больше позиции k , то медиана находится в левой части. Алгоритм рекурсивно применяется только к этому списку.
- 3 Вычисляет количество дубликатов опорного элемента в списке (вычитает из длины списка длины левой и правой частей).
- 4 Определяет, превышает ли количество дубликатов значение k :
 - а) когда это условие истинно, значит, алгоритм нашел решение, поскольку k -я позиция содержится среди дубликатов (это опорное число);
 - б) когда это условие ложно, вычитается количество дубликатов из k и рекурсивно применяется результат к правой части, которая должна содержать значение k -й позиции.

Теперь, когда вы понимаете процесс, можно взглянуть на код. Следующий пример показывает, как реализовать алгоритм быстрого выбора:

```
def quickselect(series, k):
    pivot = choice(series)

    left, right = list(), list()
    for item in series:
        if item < pivot:
```

```

        left.append(item)
    if item > pivot:
        right.append(item)
    length_left = len(left)
    if length_left > k:
        return quickselect(left, k)
    k -= length_left

    duplicates = len(series) - (length_left + len(right))
    if duplicates > k:
        return float(pivot)
    k -= duplicates

    return quickselect(right, k)

quickselect(series, 250)

12.0

```

Алгоритм эффективен, поскольку постоянно уменьшает размер задачи. Лучше всего он работает, когда случайное опорное число оказывается ближе к k -й позиции. (Правило остановки заключается в том, что опорное число — это значение в k -й позиции.) К сожалению, поскольку невозможно заранее знать k -ю позицию в неупорядоченном списке, случайный выбор с использованием равномерного распределения (когда у каждого элемента списка равные шансы быть выбранным) — это наилучшее решение, так как алгоритм в итоге находит правильное решение. Даже когда случайность не работает в пользу алгоритма, он продолжает уменьшать размер задачи, получая больше шансов найти решение, как было показано ранее в главе на примере случайного угадывания карт из колоды. По мере уменьшения колоды угадать ответ становится легче. Следующий код показывает, как использовать быстрый выбор для определения медианы списка чисел:

```

def my_median(series):
    if len(series) % 2 != 0:
        return quickselect(series, len(series)//2)
    else:
        left = quickselect(series, (len(series)-1) // 2)
        right = quickselect(series, (len(series)+1) // 2)
        return (left + right) / 2

my_median(series)

12.0

```

Моделирование методом Монте-Карло

Для понимания алгоритма быстрого выбора полезно знать, как он работает внутри. Установив счетчик внутри функции `quickselect()`, можно проверить производительность при различных условиях с помощью моделирования методом Монте-Карло:

```
def quickselect(series, k, counter=0):
    pivot = choice(series)

    left, right = list(), list()
    for item in series:
        if item < pivot:
            left.append(item)
        if item > pivot:
            right.append(item)

    counter += len(series)
    length_left = len(left)
    if length_left > k:
        return quickselect(left, k, counter)
    k -= length_left

    duplicates = series.count(pivot)
    if duplicates > k:
        return float(pivot), counter
    k -= duplicates

    return quickselect(right, k, counter)
```

Первый эксперимент пытается определить, сколько операций в среднем требуется алгоритму для нахождения медианы входного списка из 10001 числа:

```
results = list()
n = 10_001
for run in range(1, n):
    series = [randint(1, 25) for i in range(n)]
    med, count = quickselect(series, n//2)
    assert(med==median(series))
    results.append(count)

avg_ops = sum(results) / len(results)
print(f"Mean operations: {avg_ops}")

Mean operations: 27564.3649
```

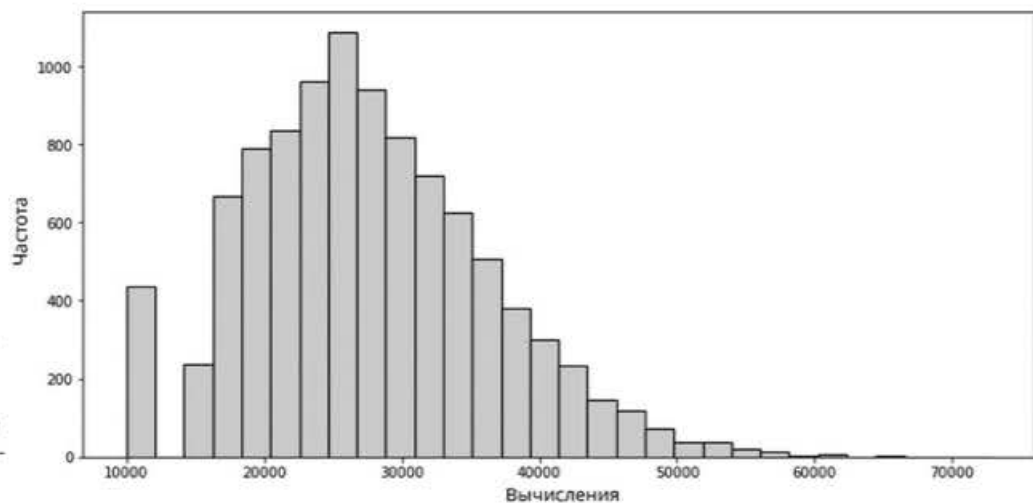

Отображение результатов на гистограмме (см. рисунок 17.4) показывает, что алгоритму требуется от двух до четырех размеров входных данных для вычислений, причем три размера — это наиболее вероятное число обработанных вычислений.

```
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 6))
plt.hist(results, bins=30, color='lightblue',
         edgcolor='black', linewidth=1.2)
plt.xlabel("Computations")
plt.ylabel("Frequency")
plt.show()
```

Если в среднем требуется около трех размеров входных данных, быстрый выбор обеспечивает хорошую производительность. Однако может возникнуть вопрос, сохранится ли пропорция между входными данными и вычислениями при увеличении размера входных данных. Как мы видели при изучении NP-полных задач, многие проблемы по-взрывному усложняются при росте размера входных данных. Эту теорию можно проверить, используя еще одно моделирование методом Монте-Карло поверх предыдущего и построив график результатов, как показано на рисунке 17.5.

РИСУНОК 17.4.
Отображение
результатов
моделирования
методом Монте-
Карло



```
input_size = [501, 1001, 5001, 10001, 20001, 50001]
computations = list()
for n in input_size:
    results = list()
    for run in range(1000):
        series = [randint(1, 25) for i in range(n)]
```

```

med, count = quickselect(series, n//2)
assert(med==median(series))
results.append(count)
avg_ops = sum(results) / len(results)
computations.append(avg_ops)

```

```

plt.figure(figsize=(12, 6))
plt.plot(input_size, computations, '-o')
plt.xlabel("Input size")
plt.ylabel("Number of computations")
plt.show()

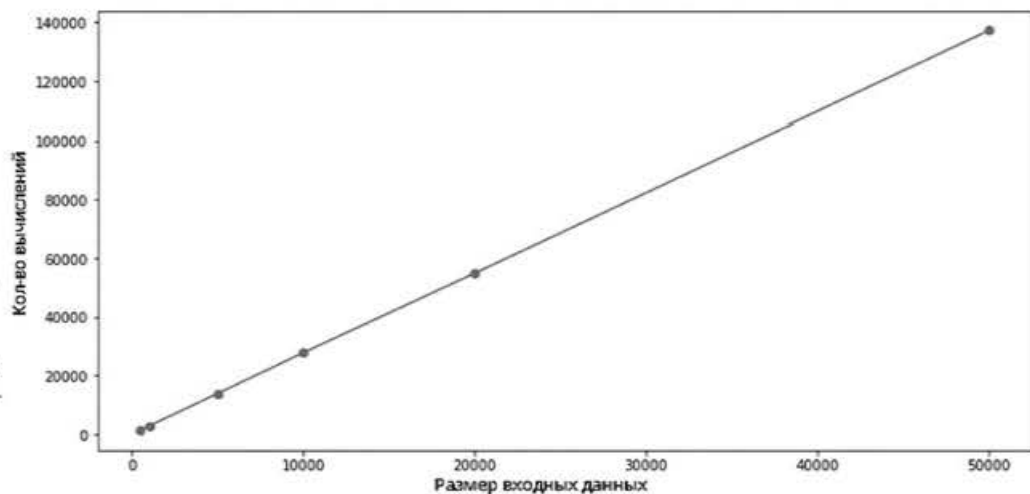
```



ВНИМАНИЕ

Выполнение вычислений для этого примера может занять до десяти минут (некоторые симуляции методом Монте-Карло могут быть довольно времязатратными), но результат помогает визуализировать, что значит работать с алгоритмом, имеющим линейное время выполнения. По мере роста входных данных (представленных по оси абсцисс) вычисления (представленные по оси ординат) растут пропорционально, что делает кривую роста идеальной прямой линией.

РИСУНОК 17.5.
Отображение
результатов
моделирования
методом Монте-
Карло при
увеличении



Более быстрая сортировка с помощью быстрой сортировки

В главе 7 объясняются алгоритмы сортировки, которые служат истинным фундаментом всех современных компьютерных алгоритмических знаний. Алгоритм быстрой сортировки, способный работать за логарифмическое время, но иногда дающий сбой и показывающий квадратичное время на плохо обусловленных входных данных, несомненно удивит вас. В этом блоке исследуются причины, по которым этот алгоритм может давать сбой, и предлагается эффективное решение путем внесения в него элемента случайности. Начнем с рассмотрения следующего кода:

```
def quicksort(series, get):

    try:
        global operations
        operations += len(series)
    except:pass
    if len(series) <= 3:
        return sorted(series)

    pivot = get(series)
    duplicates = series.count(pivot)

    left, right = list().list()
    for item in series:
        if item < pivot:
            left.append(item)
        if item > pivot:
            right.append(item)

    return quicksort(left, get) + [pivot
                                   ] * duplicates + quicksort(right, get)
```

Это еще одна реализация алгоритма из главы 7. Однако на этот раз код выделяет функцию, которая определяет опорный элемент, используемый алгоритмом для рекурсивного разделения исходного списка. Алгоритм определяет разделение, беря первое значение списка. Он также отслеживает количество операций, необходимых для завершения сортировки, используя глобальную переменную `operations`, которая определяется, сбрасывается и используется как счетчик вне функции. Следующий код тестирует алгоритм в необычных условиях, требуя от него обработки уже отсортированного списка (обратите внимание на его производительность):


```
series = list(range(25))
operations = 0
sorted_list = quicksort(series, choose_leftmost)
print(f"Operations: {operations}")
```

В этом примере `choose_leftmost()` обеспечивает базовый вывод, как показано здесь, выбирая крайний левый элемент в списке:

```
def choose_leftmost(l): return l[0]
```

Вывод показывает количество операций:

```
Operations: 322
```

В данном случае алгоритму требуется 322 операции для сортировки списка из 25 элементов, что является ужасной производительностью. Использование уже отсортированного списка вызывает проблему, поскольку алгоритм разделяет список на два списка: пустой и с остаточными значениями. Ему приходится повторять это бесполезное разделение для всех уникальных значений, присутствующих в списке. Обычно алгоритм быстрой сортировки работает хорошо, поскольку имеет дело с неупорядоченными списками, и выбор крайнего левого элемента эквивалентен случайному выбору числа в качестве опорного элемента. Чтобы избежать этой проблемы, можно использовать вариацию алгоритма, которая обеспечивает действительно случайный выбор опорного значения:

```
def choose_random(l): return choice(l)

seed(0)
series = [randint(1,25) for i in range(25)]
operations = 0
sorted_list = quicksort(series, choose_random)
print(f"Operations: {operations}")

Operations: 92
```

Теперь алгоритм выполняет свою задачу, используя немного большее количество операций, что близко к расчетному времени выполнения $n \cdot \log(n)$, то есть $25 \times \log(25) = 80,5$.

В ЭТОЙ ГЛАВЕ

- » Определение, как выполнять локальный поиск в NP-трудной задаче
- » Работа с эвристиками и соседними решениями
- » Изучение множества приемов для локального поиска
- » Решение задачи 2-SAT с помощью локального поиска и рандомизации

ГЛАВА 18

Выполнение локального поиска

Предыдущие главы показывают, что решение алгоритмов не всегда сводится к поиску точного решения из-за ограничений по времени и ресурсам или из-за сложности (а возможно, и невозможности) решаемой задачи. Такие сложные ситуации требуют компромисса — обмена точных решений на выполнимые приближения. В этой главе рассматривается подход, называемый *локальным поиском* (разновидность оптимизации), который помогает найти компромисс в сложных задачах и достичь удовлетворительных решений. Локальный поиск также иногда называют *методом локального улучшения*.

В первом параграфе главы представлены принципы работы локального поиска, а затем анализируются основные составляющие решения с его помощью. Вводятся также ключевые понятия эвристики и соседних решений, без которых локальный поиск не мог бы существовать. Наконец, вы изучите множество приемов, позволяющих улучшить работу алгоритмов локального поиска. (Рандомизация очень хорошо сочетается с большинством решений локального поиска.)

Заключительный параграф глубоко погружается в проблему 2-выполнимости (2-SAT). У этой задачи глубокие последствия в промышленном производстве (для персонализации продукции), планировании воздушного движения и составлении расписания спортивных турниров.



ЗАПОМНИТЕ

Вам не нужно вводить исходный код для этой главы вручную. На самом деле, использовать загружаемый исходный код намного проще. Исходный код для этой главы можно найти в файле **A4D2E; 18; Local Search.ipynb** из загружаемых исходников. Подробнее о том, как найти этот файл с исходным кодом, смотрите во введении.

§ 1. Понимание локального поиска

При работе с NP-трудной задачей, для которой не существует известного решения со сложностью меньше экспоненциальной (см. обсуждение теории NP-полноты в главе 15), стоит попробовать несколько альтернатив. Основываясь на идее, что задачи класса NP требуют некоторого компромисса (например, принятия частичных или неоптимальных результатов), следующие варианты предлагают решение этой в противном случае неразрешимой проблемы:

- » Определить особые случаи, при которых можно эффективно решить задачу за полиномиальное время, используя точный метод или жадный алгоритм (этот подход упрощает задачу и ограничивает количество комбинаций решений для перебора).
- » Применить методы динамического программирования (описанные в главе 16), которые улучшают полный перебор и снижают сложность задачи.
- » Пойти на компромисс и разработать приближенный алгоритм, который находит *частичное* решение, близкое к оптимальному. Когда вас устраивает частичное решение, вы сокращаете время работы алгоритма. Приближенные алгоритмы могут быть:
 - жадными алгоритмами (как обсуждалось в главе 15);
 - линейным программированием (тема главы 19);
 - выбором эвристики или *метаэвристики* (эвристики, помогающей определить, какую эвристику использовать), которая хорошо работает для вашей задачи на практике (однако у нее нет теоретических гарантий и она, как правило, основана на эмпирических данных).
- » Применить локальный поиск с использованием рандомизации или другой эвристической техники.

Локальный поиск — это общий подход к решению задач, включающий широкий спектр алгоритмов. Использование локального поиска помогает избежать экспоненциальной сложности многих NP-задач. Локальный поиск начинается с несовершенного решения проблемы и пошагово от него отходит. Он определяет жизнеспособность близлежащих решений, потенциально ведущих к идеальному решению, основываясь на случайном выборе или продуманной эвристике (точные методы не используются).



ЗАПОМНИТЕ

Эвристика — это обоснованное предположение о решении, например эмпирическое правило, которое указывает направление к желаемому результату, но не может точно сказать, как его достичь. Это похоже на ситуацию, когда вы заблудились в незнакомом городе и люди говорят вам идти определенным путем к вашему отелю (но без точных указаний) или просто сообщают, как далеко вы от него находитесь. Некоторые решения локального поиска используют эвристики, поэтому в этой главе вы найдете несколько примеров. Глава 20 подробно рассматривает использование эвристик для выполнения практических задач.

Нет гарантии, что локальный поиск приведет к решению проблемы, но ваши шансы улучшаются в сравнении с начальной точкой, если вы предоставите достаточно времени для выполнения вычислений. Поиск останавливается только тогда, когда не может найти способ улучшить достигнутое решение.

Знакомство с окрестностью

Алгоритмы локального поиска итеративно улучшают начальное решение, перемещаясь шаг за шагом через соседние решения, пока дальнейшее улучшение решения становится невозможным. Поскольку алгоритмы локального поиска так же просты и интуитивно понятны, как и жадные алгоритмы, разработать подход локального поиска к алгоритмической задаче несложно. Ключ — определить правильную процедуру:

- 1 начать с существующего решения (обычно — случайного решения или решения, полученного другим алгоритмом);
- 2 найти набор возможных новых решений в окрестности текущего решения, которые составляют *список кандидатов*;
- 3 определить, какое решение использовать вместо текущего, на основе результата эвристики, принимающей список кандидатов в качестве входных данных;
- 4 продолжить выполнение шагов 2 и 3 до тех пор, пока не перестанет наблюдаться улучшение решения, что означает достижение лучшего решения, доступного в данной окрестности.

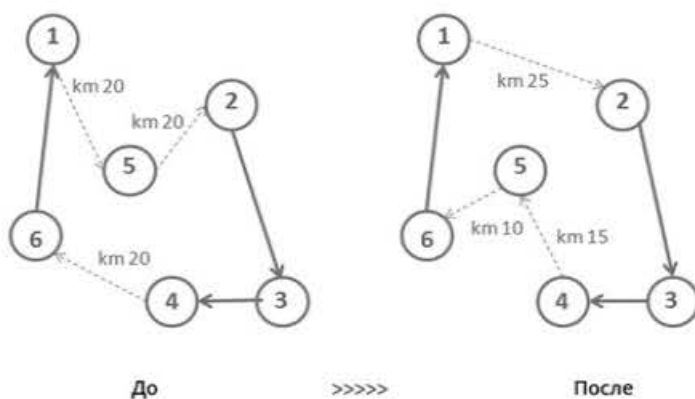


Хотя локальный поиск прост в реализации, он может не найти решение за разумное время (в этом случае можно остановить процесс через некоторое время и использовать текущее решение) или выдать решение недостаточного качества. Существуют некоторые профессиональные приемы, позволяющие получить максимальную отдачу от этого подхода.

В начале локального поиска выбирается начальное решение. Если вы решаете начать со случайного решения, полезно обернуть поиск в повторяющиеся итерации, генерируя различные случайные начальные решения. Иногда получение хорошего конечного решения зависит от начальной точки. Если вы отталкиваетесь от существующего решения, которое хотите улучшить, использование начального решения, полученного жадным алгоритмом, может оказаться хорошим компромиссом, поскольку не потребует много времени как на создание жадного решения, так и на непосредственную подгонку решения локального поиска от него.

После выбора начальной точки нужно определить окрестность и ее размер. Определение окрестности требует выяснения минимального изменения, которое можно внести в ваше решение. Если решение представляет собой набор элементов, все соседние решения — это наборы, в которых один из элементов мутирует. Например, в задаче коммивояжера (TSP) соседние решения могут включать изменение конечных городов двух (или более) маршрутов, как показано на рисунке 18.1.

РИСУНОК 18.1.
Изменение
конечных точек
маршрутов
в задаче ком-
мивояжера
может привести
к лучшим
результатам



В зависимости от того, как вы создаете окрестность, список кандидатов может быть меньше или больше. Более длинные списки требуют больше времени и вычислений, но, в отличие от коротких списков, могут предоставить больше возможностей для более раннего и качественного завершения процесса. Длина списка предполагает компромисс, который вы уточняете путем экспериментов после каждого теста, чтобы определить, приносит ли увеличение или уменьшение списка кандидатов преимущество или недостаток с точки зрения времени выполнения и качества решения.

Выбор нового решения должен основываться на эвристике, и в зависимости от задачи нужно определить лучшее решение. Например, в задаче коммивояжера используйте такие замены маршрутов, которые больше всего сокращают общую длину тура. В некоторых случаях вместо эвристики можно использовать случайный выбор (как вы узнаете в задаче SAT-2, также называемой задачей 2-SAT, в этой главе). Даже при наличии четкой эвристики алгоритм может найти несколько лучших решений. Добавление элемента случайности может сделать ваш локальный поиск эффективнее. Столкнувшись со множеством решений, можно безопасно выбрать одно из них случайным образом.



СОВЕТ

В идеале при локальном поиске лучшие результаты достигаются, когда вы запускаете несколько поисков, внося максимально возможную случайность как в начальное решение, так и в процесс выбора следующего шага. Позволяйте эвристике принимать решения только тогда, когда видите в этом явное преимущество. Локальный поиск и случайность — хорошие друзья.

Ваш поиск должен где-то остановиться, поэтому нужно выбрать правила остановки для локального поиска. Это происходит, когда эвристика больше не может найти хороших соседей или не может улучшить качество решения (например, при вычислении целевой функции, как в задаче коммивояжера, путем измерения общей длины маршрута). В зависимости от задачи, если не создать правило остановки, поиск может продолжаться бесконечно или занять неприемлемо долгое время. Если вы не можете определить точку остановки, ограничьте время поиска решений или подсчитайте количество попыток. Однако при отслеживании времени или подсчете попыток вы основываете решения на вероятности успеха алгоритма в определенной точке процесса. Хотя это удобный подход, принятие решений на основе времени или количества попыток может привести к неприемлемым решениям.

§ 2. Представляем приемы локального поиска

Локальный поиск отслеживает текущее решение и переходит к соседним решениям по одному, пока не найдет идеальное решение (или не сможет улучшить текущее решение). У него несколько ключевых преимуществ при работе с NP-трудными задачами, поскольку он:

- » прост в разработке и реализации;
- » использует мало памяти и вычислительных ресурсов (хотя поиск требует времени выполнения);
- » находит приемлемые или даже хорошие решения задачи, начиная с неидеального решения (соседние решения должны создавать путь к идеальному решению).



СОВЕТ

На задачи, которые может решить локальный поиск, можно смотреть как на граф взаимосвязанных решений. Алгоритм обходит граф, переходя от узла к узлу в поисках узла, удовлетворяющего требованиям задачи. С этой точки зрения локальный поиск использует преимущества алгоритмов исследования графов, таких как поиск в глубину (DFS) или поиск в ширину (BFS), которые обсуждаются в главе 9.

Локальный поиск предоставляет жизнеспособный способ нахождения приемлемых решений для NP-трудных задач. Однако локальный поиск не может работать правильно без подходящей эвристики. Рандомизация может хорошо сочетаться с локальным поиском и помогает, используя:

- » **случайную выборку** – генерацию начальных решений;
- » **случайное блуждание** – выбор случайного решения из числа соседних с текущим (подробнее о случайных блужданиях читайте в блоке «Решение 2-SAT с помощью рандомизации» далее в этой главе).

Рандомизация — не единственная доступная эвристика. Локальный поиск может опираться на более обоснованное исследование решений, используя целевую функцию для определения направления (как в оптимизации *восхождением к вершине*) и избегания ловушки посредственных решений (как в *имитации отжига* и *поиске с запретами*). *Целевая функция* — это вычисление, которое может оценить качество вашего решения, выдавая числовую оценку. Если при восхождении к вершине вам нужны более высокие оценки, у вас задача максимизации; если вы ищете меньшие числовые оценки — у вас задача минимизации.

Объяснение алгоритма восхождения к вершине на примере задачи о ферзях

Вы легко найдете аналогии методов, используемых в локальном поиске, поскольку многие явления подразумевают постепенный переход от одной ситуации к другой. Локальный поиск — это не просто техника, разработанная экспертами по алгоритмам, а процесс, который можно наблюдать как в природе, так и в человеческом обществе. Например, в обществе и науке инновации можно рассматривать как локальный поиск следующего шага среди доступных в настоящее время технологий: <https://www.technologyreview.com/2017/01/13/154580/mathematical-model-reveals-the-patterns-of-how-innovations-arise>. Многие эвристические методы берут начало в физическом мире, черпая вдохновение в силе тяготения, плавлении металлов, эволюции ДНК животных и поведении роев муравьев, пчел и светлячков (статья на <https://arxiv.org/pdf/1003.1464.pdf> объясняет алгоритм полета светлячков Леви и другие алгоритмы, вдохновленные природой).

Метод восхождения к вершине вдохновлен силой тяготения. Он основан на наблюдении, что, когда мяч катится вниз по долине, он выбирает самый крутой спуск, затем, встречая холм, стремится двигаться по самому прямому пути вверх, чтобы достичь вершины. Постепенно, шаг за шагом, независимо от того, поднимается он или спускается, мяч достигает своей цели — точки, где дальнейшее движение вверх или вниз невозможно.

В локальном поиске можно успешно имитировать ту же процедуру, используя *целевую функцию* — измерение, которое оценивает соседние решения и определяет, какое из них улучшает текущее. Используя аналогию с восхождением на холм, мы видим, что наличие целевой функции подобно ощущению наклона местности и определению следующего наилучшего хода. Находясь в текущей позиции, путник оценивает каждое направление, чтобы определить уклон местности. Когда цель — достичь вершины, путник выбирает направление с наибольшим подъемом. Однако это лишь идеальная ситуация; путники часто сталкиваются с проблемами во время подъема и должны использовать другие решения для их обхода.

Целевая функция похожа на жадный критерий (см. главу 5). Она слепа в отношении конечного пункта назначения, поэтому может определять направление, но не обнаруживать препятствия. Подумайте о влиянии слепоты при восхождении в горы — трудно сказать, когда путник достиг вершины. Плоская местность, не предоставляющая возможностей для движения вверх, может указывать на то, что путник достиг вершины. К сожалению, ровное место может оказаться равниной, впадиной или даже ямой, в которую угодил путник. Нельзя быть уверенным наверняка, поскольку путник не видит.

Та же проблема возникает при использовании локального поиска с эвристикой восхождения к вершине: алгоритм последовательно находит все лучшие соседние решения, пока не достигнет точки, где среди вариантов, окружающих текущее решение, нет ничего лучше. Тут алгоритм объявляет, что решение найдено. Он также утверждает, что нашел глобальное решение, хотя, как показано на рисунке 18.2, это может быть всего лишь локальный максимум — решение, которое лучше окружающих просто потому, что вокруг него находятся худшие варианты. При дальнейшем поиске вполне возможно найти решение получше.

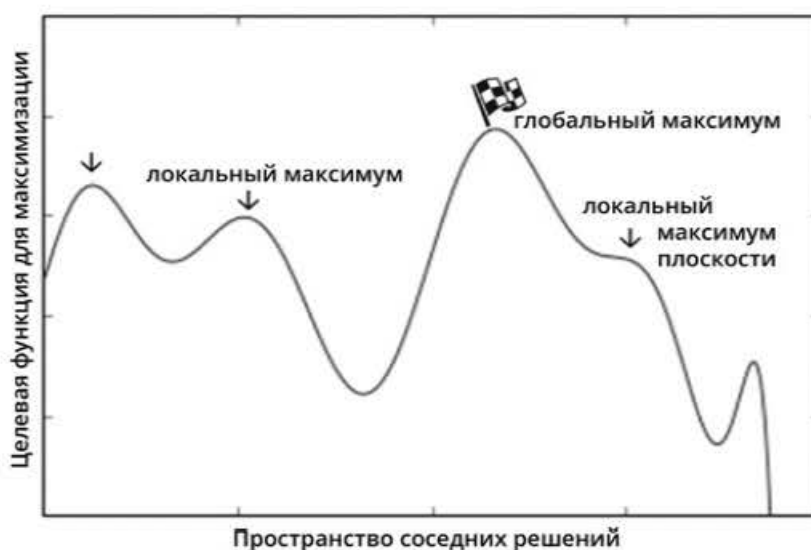


РИСУНОК 18.2.
Локальный поиск исследует ландшафт методом восхождения к вершине

Хороший пример работы метода восхождения (и рисков застрять в локальном максимуме или минимуме при спуске, как в данном примере) — головоломка с n ферзями, впервые предложенная шахматным экспертом Максом Беццелем в 1848 году как вызов для любителей шахмат. В этой задаче нужно разместить определенное количество ферзей (это число n) на шахматной доске размером $n \times n$. Их нужно расставить так, чтобы ни один ферзь не находился под ударом любого другого. (В шахматах ферзь может атаковать по любому направлению — по горизонтали, вертикали или диагонали.)



СОВЕТ

Это действительно NP-трудная задача. Если нужно расставить восемь ферзей на доске 8×8 , существует 4426 165 368 различных способов их размещения, но только 92 конфигурации решают задачу. Очевидно, что решить эту проблему методом полного перебора или просто наугад невозможно. Локальный поиск решает эту задачу очень простым способом, используя метод восхождения:

- 1 случайным образом размещает n ферзей на шахматной доске так, чтобы каждый находился в отдельном столбце (не более одного ферзя в столбце);
- 2 оценивает следующий набор решений, перемещая каждого ферзя на одну клетку вверх или вниз в пределах его столбца (этот шаг требует $2 \times n$ ходов);
- 3 определяет, сколько ферзей атакует друг друга после каждого хода;
- 4 определяет, у какого решения наименьшее количество атакующих друг друга ферзей, и использует это решение как отправную точку для следующей итерации;
- 5 повторяет шаги со 2 по 4, пока не будет найдено решение.

К сожалению, этот подход работает только примерно в 14% случаев на стандартной доске 8×8 , поскольку в 86% случаев алгоритм застревает в такой конфигурации шахматной доски, которая не допускает дальнейших улучшений. (Количество атакуемых ферзей не уменьшается для всех $2 \times n$ доступных ходов в качестве следующих решений.) Единственный способ выйти из такого тупика — перезапустить локальный поиск с нуля, выбрав другую случайную начальную расстановку ферзей на шахматной доске. В среднем настойчивость будет вознаграждена в одной попытке из десяти, что менее эффективно, чем систематический перебор всех возможных решений. На рисунке 18.3 показано успешное решение.

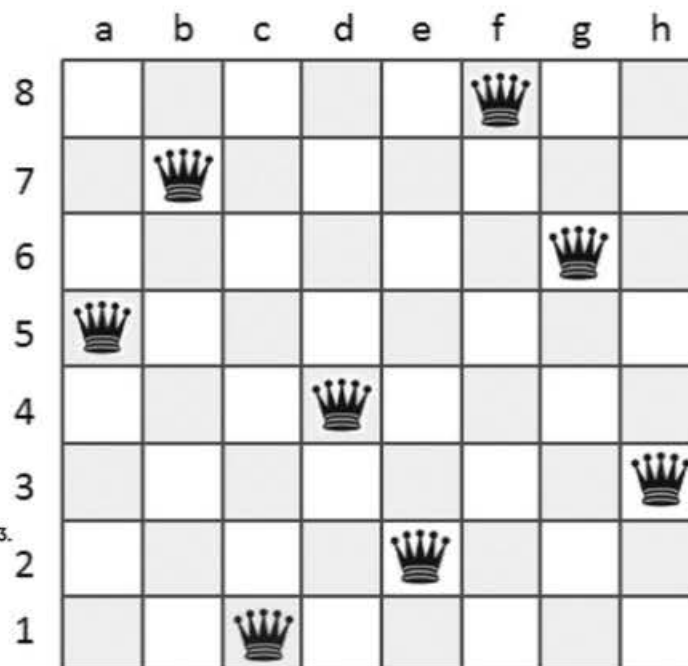


РИСУНОК 18.3.
Решение
головоломки
с восемью
ферзями

Несмотря на свою слабость в нахождении наилучшего решения (или решения вообще) с первой попытки, алгоритмы восхождения к вершине не используются повсеместно, особенно в искусственном интеллекте и машинном обучении. Нейронные сети, которые распознают звуки или изображения, работают в мобильных телефонах и управляют беспилотными автомобилями, в основном полагаясь на оптимизацию методом восхождения, называемую *градиентным спуском*. Рандомизированные старты и случайные включения в процедуру восхождения позволяют избежать локальных решений и достичь глобального максимума. И имитация отжига и поиск с запретами — это разумные способы использования случайных решений в методе восхождения.

Знакомство с имитацией отжига

В определенное время поиска, если целевая функция перестает давать правильные указания, можно использовать другую эвристику для контроля ситуации и попытаться найти лучший путь к лучшему решению задачи. Именно так работают и имитация отжига, и поиск с запретами: они обеспечивают аварийный выход, когда это необходимо.

Имитация отжига получила свое название от металлургической техники, при которой металл нагревают до высокой температуры, а затем медленно охлаждают, чтобы размягчить его для холодной обработки и устранить внутренние кристаллические дефекты (подробнее об этом металлургическом процессе см. на <https://www.azom.com/article.aspx?ArticleID=20195>). Локальный поиск воспроизводит эту идею, рассматривая поиск решения как атомную структуру, которая изменяется для улучшения своей работоспособности. Температура — переломный этап в процессе оптимизации. Подобно тому, как высокие температуры заставляют структуру материала расслабляться (твердые тела плавятся, а жидкости испаряются при высоких температурах), высокие температуры в алгоритме локального поиска вызывают релаксацию целевой функции, позволяя ей предпочитать худшие решения лучшим. Имитация отжига модифицирует процедуру восхождения, сохраняя целевую функцию для оценки соседних решений, но позволяя ей определять выбор решения поиска иным образом, а именно:

- 1 получить температуру, выраженную как вероятность (физическая функция Гиббса — Больцмана является формулой, которая преобразует температуру в вероятность, но можно использовать и более простые подходы, сопоставляющие температуру с вероятностью);

- 2** установить график изменения температуры (температура уменьшается с определенной скоростью по мере течения времени и выполнения поиска);
- 3** определить начальное решение (используя случайную выборку или другой алгоритм) и запустить цикл (по мере выполнения цикла температура снижается);
- 4** остановить оптимизацию, когда температура достигнет нуля;
- 5** предложить текущий результат в качестве решения.

На этом этапе нужно выполнить итеративный поиск решений. Для каждого шага в предыдущей итерации между шагами 3 и 4 выполните следующее:

- 1** составьте список соседних решений и выберите одно из них случайным образом;
- 2** установите соседнее решение как текущее, если оно лучше текущего;
- 3** в противном случае выберите случайное число от 0 до 1 на основе пороговой вероятности, связанной с текущей температурой, и определите, меньше ли оно пороговой вероятности:
 - если меньше – установите соседнее решение как текущее (даже если оно хуже текущего решения согласно целевой функции);
 - если больше – сохраните текущее решение.

Имитация отжига — это умный способ улучшить метод восхождения к вершине, поскольку позволяет избежать остановки поиска на локальном решении. Когда температура достаточно высока, поиск может использовать случайное решение и найти другой путь к лучшей оптимизации. Поскольку температура выше в начале поиска, у алгоритма есть шанс внести случайность в оптимизацию. По мере охлаждения температуры до нуля вероятность выбора случайного решения становится все меньше и меньше и локальный поиск продолжается как при восхождении к вершине. Например, в задаче коммивояжера алгоритм осуществляет имитацию отжига, проверяя текущее решение при высоких температурах путем:

- » случайного выбора сегмента маршрута и прохождения его в противоположном направлении;
- » посещения города раньше или позже в маршруте, сохраняя при этом порядок посещения других городов.

Если полученные корректировки ухудшают длину маршрута, алгоритм сохраняет или отвергает их в соответствии с температурой в процессе имитации отжига.

Избегание повторов с помощью поиска с запретами

Слово *табу* происходит из полинезийского языка тонга и означает, что к определенным вещам нельзя прикасаться, поскольку они священны. Слово «табу» (которое пишется как *taboo* в английском языке) перешло из антропологических исследований в повседневный язык для обозначения чего-то запрещенного.

При локальной оптимизации поиска часто можно застрять в окрестности решений, которые не предлагают никакого улучшения, то есть это локальное решение, которое кажется лучшим, но оно далеко от желаемого решения. Поиск с запретами ослабляет одни правила и усиливает другие, чтобы предложить выход из локальных минимумов и помочь достичь лучших решений.

Эвристический метод поиска с запретами оперирует целевыми функциями и продвигается через множество соседних решений. Он вмешивается, когда дальнейшее продвижение невозможно из-за того, что следующие решения не улучшают целевой функции. В таких случаях поиск с запретами выполняет следующие действия:

- » временно допускает использование худшего решения, чтобы проверить, поможет ли отход от локального решения найти лучший путь к оптимальному результату;
- » запоминает опробованные решения и запрещает их повторное использование, тем самым гарантируя, что поиск не заикнется между одними и теми же решениями вокруг локального оптимума, не находя выхода;
- » создает долгосрочную или краткосрочную память запрещенных решений, изменяя длину очереди для хранения предыдущих решений (когда очередь заполняется, эвристика удаляет самый старый запрет, чтобы освободить место для нового).

Поиск с запретами можно сравнить с кешированием и мемоизацией (см. главу 16), поскольку алгоритму нужно отслеживать свои шаги, чтобы экономить время и избегать повторного перебора ранее использованных решений. В задаче коммивояжера это помогает при оптимизации решения путем изменения порядка посещения двух или более городов, позволяя избежать повторения наборов решений.

§ 3. Решение выполнимости булевых схем

В качестве практического примера работы локального поиска в этом параграфе рассматривается задача выполнимости схемы — классическая NP-полная проблема со множеством практических применений, включая промышленное планирование, составление спортивных расписаний и управление воздушным движением. Алгоритм используется и для тестирования работы электронных схем, оптимизируя их путем удаления цепей, которые не будут проводить электрические сигналы (они избыточны). Более того, алгоритм решения находит применение во многих других областях: автоматической маркировке на картах и диаграммах, дискретной томографии, планировании с ограничениями, кластеризации данных и других задачах, где нужно принимать противоречивые решения. Он использует подход, основанный на рандомизации и алгоритме Монте-Карло. Как было рассмотрено в главе 17, алгоритм Монте-Карло опирается на случайный выбор в процессе оптимизации и не гарантирует успешного выполнения задачи, хотя имеет высокую вероятность успешного завершения.

Пример посвящен тестированию компьютерных схем. Компьютерные схемы состоят из серии соединенных компонентов, каждый из которых открывает или закрывает цепь в зависимости от входных сигналов. Такие элементы называются *логическими вентилями* (физически их роль выполняют транзисторы), и, если вы создаете схему со множеством логических вентилях, нужно понимать, может ли через нее проходить электричество и при каких условиях.

В главе 14 рассматривается внутреннее представление компьютера, основанное на нулях (отсутствие электричества в цепи) и единицах (наличие электричества). Это представление 0/1 можно рассматривать с логической точки зрения, преобразуя сигналы в условия *False* (нет электричества в цепи) или *True* (электричество присутствует). В главе 4 рассматриваются логические операторы (**AND**, **OR** и **NOT**), как показано на рисунке 18.4, которые взаимодействуют с входными значениями *True* и *False*, преобразуя их в различные выходные значения. Все эти концепции помогают представить физическую электрическую цепь как последовательность логических операторов, определяющих логические вентили. Комбинация всех их условий определяет, может ли цепь проводить электричество.

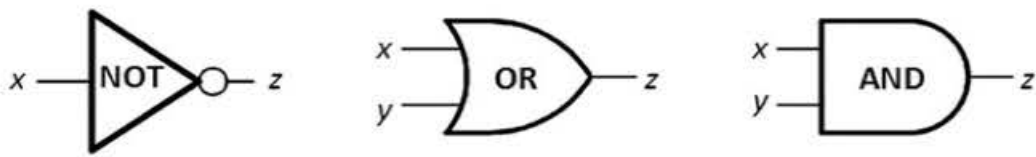


РИСУНОК 18.4.
Символы и таблицы истинности логических операторов AND, OR и NOT

Input		Output
x	z	
0	1	
1	0	

Input		Output
x	y	z
0	0	0
0	1	1
1	0	1
1	1	1

Input		Output
x	y	z
0	0	0
0	1	0
1	0	0
1	1	1

Такое логическое представление называется *комбинационной булевой схемой*, а тест для проверки ее функциональности — *задачей выполнимости схемы*. В простейшем случае схема состоит только из условий **NOT** (называемых *инверторами*), принимающих один проводной вход, и условий **OR**, принимающих два провода в качестве входов. Это сценарий 2-выполнимости (2-SAT), и если алгоритм будет решать его методом полного перебора, то потребуется в худшем случае $2k$ проверок (где k — количество входных проводов), чтобы найти набор условий, при которых электричество пройдет через всю схему, если такой набор существует. Есть и более сложные версии задачи, допускающие больше входов для каждого логического вентиля **OR** и использующие вентили **AND**, но они выходят за рамки этой книги.

Решение 2-SAT с помощью рандомизации

Независимо от того, какую электронную схему вам нужно протестировать с помощью булева представления, вы можете представить ее в виде вектора булевых переменных. Вы также можете создать другой вектор для хранения *клауз* — набора условий, которым должна удовлетворять схема (например, что провод *A* и провод *B* должны быть оба **True**). Это не единственный способ представления задачи; фактически существуют другие решения с использованием графов. Однако для этого примера достаточно этих двух векторов.

Задачу можно решить с помощью рандомизированного локального поиска за полиномиальное время. Профессор Христос Пападимитриу, преподающий в Калифорнийском университете в Беркли (<https://www.engineering.columbia.edu/faculty/christos-papadimitriou>), разработал алгоритм, названный *RandomWalkSAT*. Он представил его в статье «О выборе удовлетворяющего присваивания значений истинности», опубликованной в 1991 году в материалах 32-го симпозиума IEEE по основам

информатики. Алгоритм конкурентоспособен в сравнении с более обоснованными методами и служит показательным примером локального поиска, поскольку вносит только одно изменение за раз в текущее решение. Он использует два вложенных цикла: один для многократных попыток с начальным решением, а другой — для случайного изменения начального случайного решения. Внешний цикл повторяется $\log_2(k)$ раз (где k — количество проводов). Внутренний цикл включает следующие шаги:

- 1 выбрать случайное решение задачи;
- 2 повторить следующие шаги $2 \times k^2$ раз:
 - а) определить, является ли текущее решение правильным (если это верное решение — прервать все циклы и сообщить о решении);
 - б) случайным образом выбрать неудовлетворенное условие (случайно выбрать одно из условий в нем и изменить его).

Реализация кода на Python

Чтобы решить задачу 2-SAT с помощью Python и алгоритма RandomWalkSAT, нужно определить несколько вспомогательных функций. Функции `create_clauses()` и `signed()` помогают сгенерировать задачу для схемы, обрабатывая логические элементы **OR** и **NOT** соответственно. Используя эти функции, вы указываете количество элементов **OR** и задаете начальное число, которое гарантирует возможность воссоздать полученную задачу позже (позволяя пробовать решить задачу многократно и на разных компьютерах).

Функция `create_random_solutions()` обеспечивает холодный старт задачи, предоставляя случайное решение, которое устанавливает входные значения. Вероятность найти правильное решение случайным образом крайне мала (одна из двух в степени количества элементов), но в среднем можно ожидать, что три четверти элементов установлены правильно (поскольку, как видно из таблицы истинности для функции **OR**, три входа из четырех возможных дают **True**). Функция `check_solution()` определяет, когда схема удовлетворена (указывая на правильное решение). В противном случае она выводит неудовлетворенные условия. (Этот код можно найти в загружаемом файле исходного кода [A4D2E; 18; Local Search.ipynb](#); подробнее см. во введении.)

```
import random
from math import log2
```

```

def signed(v):
    return v if random.random() < 0.5 else -v

def create_clauses(i, seed=1):
    random.seed(seed)
    return [(signed(random.randint(0, i-1)),
                signed(random.randint(0, i-1)))
            for j in range(i)]

def create_random_solution(i, *kwargs):
    return {j:signed(1)==1 for j in range(i)}

def check_solution(solution, clauses):
    violations = list()
    for k,(a,b) in enumerate(clauses):
        if not (((solution[abs(a)]) == (a > 0)) |
                ((solution[abs(b)]) == (b > 0))):
            violations.append(k)
    return violations

```

После определения этих функций у вас есть все строительные блоки для функции `sat2()`, решающей задачу. Это решение использует две вложенные итерации: первая повторяет множество стартов, вторая случайным образом выбирает неудовлетворенные условия и делает их истинными. Решение выполняется за полиномиальное время. Функция не гарантирует нахождения решения, если оно существует, но вероятность того, что она найдет решение при его наличии, высока. Фактически внутренний итерационный цикл делает $2 \times k^2$ случайных попыток решить схему, чего обычно достаточно для случайного блуждания по линии, чтобы достичь цели.



ЗАПОМНИТЕ

Случайное блуждание — это серия вычислений, представляющих объект, который удаляется от своего начального положения, выбирая случайное направление на каждом шаге. Случайное блуждание можно представить как путешествие пьяного человека от одного фонарного столба к другому. Случайные блуждания полезны для представления математической модели многих реальных аспектов. Они находят применение в биологии, физике, химии, информатике и экономике, особенно в анализе фондового рынка. Если хотите узнать больше о случайных блужданиях, перейдите по ссылке [http://www.mit.edu/~kardar/teaching/projects/chemotaxis\(AndreaSchmidt\)/random.htm](http://www.mit.edu/~kardar/teaching/projects/chemotaxis(AndreaSchmidt)/random.htm).

Случайное блуждание по прямой — это простейший пример случайного блуждания. В среднем требуется k^2 шагов случайного блуждания, чтобы достичь расстояния k от начальной точки. Это ожидаемое усилие объясняет, почему алгоритму RandomWalkSAT требуется $2 \times k^2$ случайных

попыток для исправления начального решения. Такое количество попыток обеспечивает высокую вероятность того, что алгоритм исправит k условий. Более того, он работает так же, как игра в случайное угадывание карт, рассмотренная в предыдущей главе. По мере выполнения алгоритма выбор правильного ответа становится проще. Внешние повторения гарантируют выход из неудачных случайных выборов во внутреннем цикле, которые могут остановить процесс на локальном решении.

```
def sat2(clauses, n, start=create_random_solution):
    not_solved = True
    for external_loop in range(round(log2(n))):
        solution = start(n, clauses)
        history = list()
        for internal_loop in range(2 * n**2):
            response = check_solution(solution, clauses)
            unsatisfied = len(response)
            history.append(unsatisfied)
            if unsatisfied==0:
                print ("Solution in %i external loops," %
                        (external_loop + 1), end=" ")
                print ("%i internal loops" %
                        (internal_loop + 1))
                not_solved = False
                break
            else:
                r1 = random.choice(response)
                r2 = random.randint(0, 1)
                clause_to_fix = clauses[r1][r2]
                solution[abs(clause_to_fix)] = (
                    clause_to_fix>0)
        else:
            continue
        break
    if not_solved is True:
        print("Not solvable")
    return history, solution
```

Теперь, когда все функции настроены правильно, можно запустить код для решения задачи. Вот первый пример, который проверяет схему, созданную с начальным значением 0 и использующую 1000 логических вентилей:

```
n = 1000
clauses = create_clauses(n, seed=0)
history, solution = sat2(clauses, n,
                        start=create_random_solution)
```


Found solution in 1 external loops, 674 internal loops

Если построить решение в виде графика, где по оси абсцисс отложено количество шагов (случайных исправлений решения), а по оси ординат — количество оставшихся для исправления условий, можно убедиться, что алгоритм в долгосрочной перспективе находит правильное решение, как показано на рисунке 18.5.

```
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 6))
plt.plot(history, 'b-')
plt.xlabel("Random adjustments")
plt.ylabel("Unsatisfied clauses")
plt.grid(True)
plt.show()
```

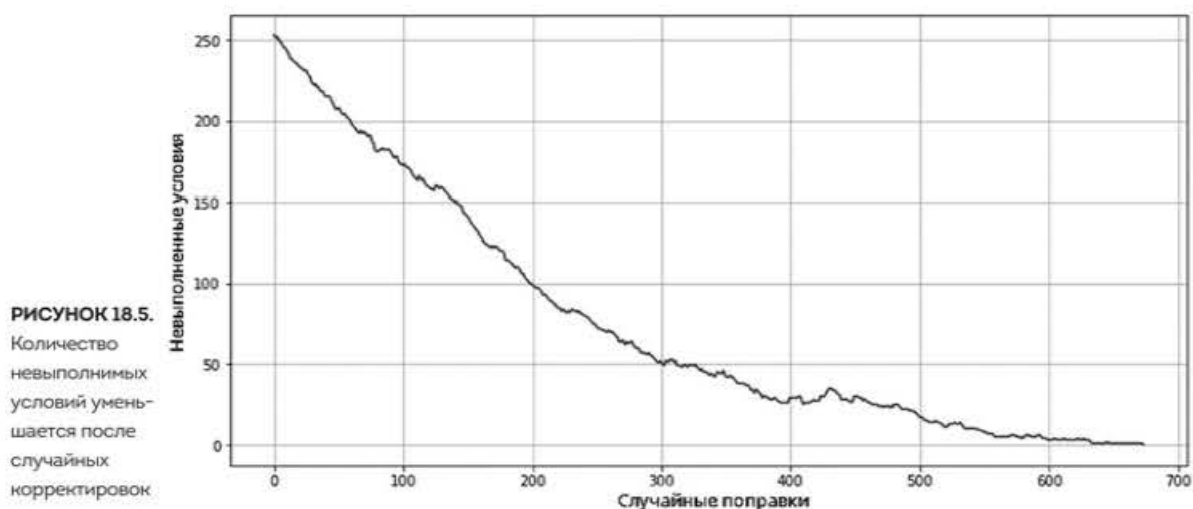


РИСУНОК 18.5.
Количество невыполнимых условий уменьшается после случайных корректировок

Не все условия фактически разрешимы. Вы можете попробовать разные начальные значения и найти такие, обработка которых занимает много времени, например схему с 1000 вентилями, которую можно создать, установив начальное значение равным 12. Когда вы создаете неразрешимую схему, скрипт возвращает сообщение о том, что схема неразрешима после выполнения нужного количества попыток. Когда встречается неразрешимый набор условий, это занимает больше времени. Это происходит из-за внешнего цикла, которому нужно исчерпать эквивалент $\log_2(k)$ от количества условий, чтобы быть статистически уверенным в том, что решение не находится потому, что схема не работает, а не потому, что вам просто не повезло в рандомизированном локальном поиске.

Осознание важности начальной точки

Хотя у алгоритма RandomWalkSAT есть временная сложность $O(\log 2k \cdot k^2)$ в худшем случае, где k — количество входных данных, его можно ускорить, подобрав начальную точку. Действительно, хотя при старте со случайной конфигурации в среднем четверть условий остается невыполненной, многие из них можно исправить, выполнив проход по данным.

Проблема с условиями заключается в том, что многие требуют значения `True` на входе, а одновременно многие другие требуют значения `False`. Когда все условия требуют, чтобы вход был `True` или `False`, вы можете установить его в требуемое состояние, что удовлетворяет большому количеству условий и упрощает решение оставшихся. Следующая новая реализация алгоритма RandomWalkSAT включает начальную фазу, которая сразу решает ситуации, когда вход требует определенного значения `True` или `False` для всех условий, с которыми он взаимодействует:

```
def better_start(n, clauses):
    clause_dict = dict()
    for pair in clauses:
        for clause in pair:
            if abs(clause) in clause_dict:
                clause_dict[abs(clause)].add(clause)
            else:
                clause_dict[abs(clause)] = {clause}

    solution = create_random_solution(n)

    for clause, value in clause_dict.items():
        if len(value) == 1:
            solution[clause] = value.pop() > 0
    return solution
```

Код определяет новую функцию для холодного старта, которая после генерации случайного решения просматривает его и находит все входы, связанные с одним состоянием (`True` или `False`). Устанавливая их сразу в требуемое состояние, можно уменьшить количество условий, требующих изменения, что позволяет локальному поиску выполнить меньше работы и завершиться раньше:

$n = 1000$

```
clauses = create_clauses(n, seed=0)
history, solution = sat2(clauses, n, start=better_start)
Found solution in 1 external loops, 400 internal loops
```

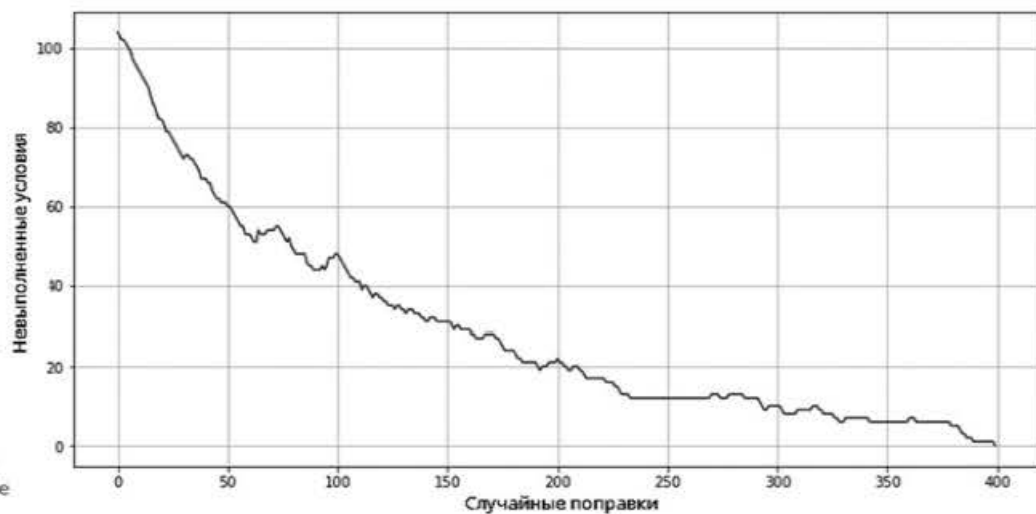
После построения графика результатов сразу видно улучшение при использовании этой новой, упрощенной начальной точки. В среднем для выполнения задачи требуется меньше операций.



СОВЕТ

В локальном поиске всегда помните, что начальная точка важна для более раннего и успешного завершения алгоритма, как показано на рисунке 18.6. В целом старайтесь обеспечить максимально качественное начало для вашего поиска.

РИСУНОК 18.6. Выполнение происходит быстрее благодаря лучшей начальной точке



- » Как выполняется оптимизация с помощью линейного программирования
- » Преобразование задач из реального мира в математические и геометрические
- » Использование Python для решения задач линейного программирования

ГЛАВА 19

Применение линейного программирования

Линейное программирование (LP) также называется линейной оптимизацией и представляет собой метод поиска максимального или минимального значения функции, называемой целевой. Целевая функция ограничена некоторыми границами, также называемыми ограничениями. Как следует из названия, LP работает, опираясь на функции и ограничения, основанные на линейности, где все может быть представлено в виде линии на графе; никаких кривых не допускается. Несмотря на такие ограничения, если вы можете упростить и определить вашу задачу на основе линейных функций, LP служит мощным инструментом. Его применение обширно и важно, затрагивает повседневные задачи в производстве и логистике.

Эта глава поможет вам понять линейное программирование. Первый параграф знакомит вас с концепциями, необходимыми для эффективного использования линейного программирования. Он включает базовую математику, которую нельзя избежать при реализации LP. В первом параграфе также показано, как более простые задачи можно визуально представить графически на диаграмме.

Во втором параграфе вы узнаете, как применять линейное программирование к реальным задачам, используя Python в качестве инструмента для выражения этих задач в коде. Здесь вы познакомитесь с пакетом PuLP для Python, который прост как в использовании, так и в применении к вашим собственным задачам. Использование этого пакета дает

вам возможность решать задачи, которые, хотя и остаются линейными, настолько сложны, что их невозможно представить с помощью бумаги и карандаша на графике.



ЗАПОМНИТЕ

Вам не нужно вручную набирать исходный код для этой главы. На самом деле, использовать загружаемый исходный код намного проще. Исходный код главы можно найти в файле `A4D2E; 19; Linear Programming.ipynb` из загружаемых материалов. Подробнее о том, как найти этот файл, смотрите во введении.

§ 1. Использование линейных функций в качестве инструмента

Линейное программирование впервые появилось во время Второй мировой войны, когда логистика оказалась критически важной для перемещения армий, насчитывающих миллионы солдат, вооружения и припасов по географически разнообразным полям сражений. Танкам и самолетам нужны были дозаправка и перевооружение, что требовало масштабных организационных усилий для успеха, несмотря на ограничения по времени, ресурсам и действия противника.

Большинство этих военных задач можно выразить в математической форме. Математик Джордж Бернард Данциг, работавший в Управлении статистического контроля ВВС США, разработал умный способ решения этих задач с помощью *симплекс-алгоритма*. Симплекс-метод стал ключевой идеей, которая после войны пробудила интерес к численной оптимизации и положила начало многообещающей области линейного программирования. Появление первых полезных компьютеров того времени тоже усилило интерес, сделав возможным решение сложных вычислений новым и быстрым способом. Можно рассматривать раннюю историю вычислительной техники 1950-х и 1960-х годов как поиск путей оптимизации логистических задач с использованием симплекс-метода и применением как высокоскоростных компьютеров, так и специализированных языков программирования.

Данциг умер в 2005 году, но область, которую он основал, продолжает постоянно развиваться. В последние годы успешно появляются новые идеи и методы, связанные с линейным программированием, такие как:

- » **программирование в ограничениях** – выражает отношения между переменными в компьютерной программе как ограничения в линейном программировании;

» **генетические алгоритмы** — основаны на идее, что математические формулы могут реплицироваться и мутировать для решения задач так же, как это делает ДНК в природе путем эволюции. (Генетические алгоритмы также рассматриваются в главе 20 из-за их эвристического подхода к оптимизации.)

Чтобы начать работу с линейным программированием, нужно сначала понять его особый подход и терминологию, отличающиеся от рассмотренных ранее для других решений. В этом блоке показано, как подойти к задаче, в которой цель и ограничения преобразованы в линейные функции. *Целевая функция* задачи представляет собой выражение затрат, прибыли или другой величины, которую нужно максимизировать или минимизировать с учетом ограничений. *Ограничения* — это линейные неравенства, вытекающие из условий задачи, например ограничение в 40 часов рабочей недели. Цель линейного программирования — найти оптимальное числовое решение, которое может быть максимальным или минимальным значением, и определить набор условий для его получения.

Такое определение линейного программирования может показаться несколько сложным, поскольку включает математику и некоторую абстракцию (целевая функция и ограничения как линейные функции), но все станет яснее, когда вы понимаете, что такое функция и как определить, является ли она линейной. За математической терминологией линейное программирование — это просто другой взгляд на алгоритмические задачи: вместо операций и манипуляций с входными данными вы работаете с математическими функциями и выполняете вычисления с помощью специальной программы, называемой *оптимизатором*.

Линейное программирование подходит не для всех задач, но большое их количество соответствует его требованиям, особенно задачи, требующие оптимизации с учетом заранее определенных ограничений. В предыдущих главах обсуждалось, что динамическое программирование — это лучший подход, когда нужно оптимизировать задачи с ограничениями. Динамическое программирование работает с дискретными задачами, то есть с целыми числами. Линейное программирование в основном работает с десятичными числами, хотя существуют специальные алгоритмы оптимизации, предоставляющие решения в виде целых чисел (например, задачу коммивояжера можно решить с помощью целочисленного линейного программирования). У линейного программирования более широкая область применения, поскольку оно может справляться практически с любой задачей, решаемой за полиномиальное время.

Линейное программирование используется в производстве, логистике, транспорте (особенно в авиакомпаниях для определения маршрутов, расписаний и стоимости билетов), маркетинге, финансах и телеком-

муникации. Все эти приложения требуют получения максимального экономического результата при минимальных затратах с оптимизацией распределения доступных ресурсов и удовлетворением всех ограничений и лимитов. Кроме того, линейное программирование можно применять в таких распространенных приложениях, как видеоигры и компьютерная визуализация, поскольку игры основаны на двумерных и трехмерных сложных фигурах и нужно определять их столкновения, а также обеспечивать соблюдение правил игры. Эти цели достигаются с помощью алгоритма построения выпуклой оболочки, основанного на линейном программировании (см. на <https://0fps.net/category/programming/video-games>). Наконец, линейное программирование применяется в поисковых системах для задач нахождения документов. Вы можете преобразовать слова, фразы и документы в функции и определить, как максимизировать результаты поиска (получить документы, необходимые для ответа на ваш запрос), когда ищете документы с определенными математическими характеристиками.

Осваиваем необходимую математику

В программировании функции предоставляют средства для упаковки кода, который вы планируете использовать более одного раза. Функции превращают код в черный ящик — сущность, которой вы предоставляете входные данные, ожидая определенные выходные результаты (в главе 4 рассказывается, как создавать функции в Python). Математика использует функции аналогично программированию: это набор математических операций, преобразующих некоторые входные данные в выходные. Входные данные могут включать одну или несколько переменных, в результате чего получается уникальный выходной результат на основе входных данных. Обычно функция имеет следующую форму:

$$f(x) = x * 2$$

где:

- » f — определяет имя функции (оно может быть любым — можно использовать любую букву алфавита или даже слово);
- » (x) — задает входные данные (в этом примере входными данными является переменная x , но вы можете использовать больше входных данных любой сложности, включая множественные переменные или матрицы);
- » $x * 2$ — определяет набор операций, которые функция выполняет после получения входных данных (результат — выходное значение функции в виде числа).

Если в этом примере подставить входное значение 2 в качестве x , получится:

$$f(2) = 4$$

В математических терминах, вызывая эту функцию, вы отображали входное значение 2 в выходное значение 4.



ЗАПОМНИТЕ

Функции могут быть простыми или сложными, но у каждой — один, и только один, результат (даже если этот результат — кортеж) для каждого набора входных данных, которые вы предоставляете (даже когда входные данные состоят из нескольких переменных).

Линейное программирование использует функции для математического представления целей, которые нужно достичь, чтобы решить поставленную задачу. Когда вы преобразуете цели в математическую функцию, задача сводится к определению входных данных для функции, которые отображаются в максимальный выходной результат (или минимальный, в зависимости от того, чего вы хотите достичь). Функция, представляющая цель оптимизации, называется *целевой*. Кроме того, линейное программирование использует функции и неравенства для выражения ограничений или границ, которые не позволяют подставлять любые входные данные в целевую функцию. Вот, например, неравенства:

$$\begin{aligned} 0 &\geq x \leq 4 \\ y + x &< 10 \end{aligned}$$

Первое из этих неравенств означает ограничение входных данных целевой функции значениями от 0 до 4. Неравенства могут одновременно включать более одной входной переменной. Второе неравенство связывает значения одного входного параметра с другими значениями, поскольку их сумма не может превышать 10.



ЗАПОМНИТЕ

Границы подразумевают ограничение на значение входных данных, как в первом примере. *Ограничения* всегда включают математическое выражение, содержащее более одной переменной, как во втором примере.

Последнее требование линейного программирования заключается в том, что и целевая функция, и неравенства должны быть линейными выражениями. Это означает, что целевая функция и неравенства не могут содержать перемножающиеся переменные или переменные в степени (например, в квадрате или кубе).



ПРИМЕЧАНИЕ

Все функции в оптимизации должны быть линейными выражениями, поскольку процедура представляет их как линии в декартовом пространстве. (Если вам нужно освежить понятие декартова пространства, полезную

информацию можно найти по адресу <https://www.mathsisfun.com/data/cartesian-coordinates.html>.) Как объясняется в параграфе «Использование линейного программирования на практике» далее в этой главе, работу с линейным программированием можно представить скорее как решение геометрической задачи, чем математической.

Учимся упрощать при планировании

Задачи, которые решал исходный симплекс-алгоритм, были похожи на те, что обычно встречаются в учебниках в виде текстовых задач. В таких задачах все данные, информация и ограничения сформулированы четко, нет лишней или избыточной информации и для решения явно требуется применить математическую формулу (причем, скорее всего, ту, которую вы только что изучили).

В реальном мире решения проблем никогда не бывают так аккуратно изложены. Вместо этого они часто предстают в запутанном виде, а некоторая необходимая информация не сразу доступна для обработки. Тем не менее вы можете проанализировать проблему и найти нужные данные и другую информацию. Кроме того, вы можете обнаружить ограничения, такие как деньги, время или какие-то правила или распоряжения, которые нужно учитывать. При решении проблемы вы собираете информацию и разрабатываете способы ее упрощения.

Упрощение подразумевает некоторую потерю реалистичности, но делает описание проблемы более простым, что может высветить лежащие в основе процессы, приводящие всё в движение, тем самым помогая вам принять решение о том, что происходит. Более простая задача позволяет разработать модель, представляющую реальность. Модель может приближенно отражать то, что происходит в реальности, и вы можете использовать ее как для управления симуляциями, так и для линейного программирования.

Например, если вы работаете на фабрике и должны спланировать производственный график, то знаете, что чем больше людей вы добавите, тем быстрее будет производство. Однако вы не всегда получите одинаковый прирост при одинаковом добавлении людей. Например, на результаты влияют навыки операторов, которых вы привлекаете к работе. Кроме того, вы можете обнаружить, что привлечение большего количества людей к работе приносит убывающие результаты, когда эти люди тратят больше времени на общение и координацию между собой, чем на выполнение полезной работы. Тем не менее вы можете упростить модель, предположив, что каждый человек, которого вы привлекаете к задаче, будет производить определенное количество конечных или промежуточных товаров.

Работа с геометрией с помощью симплекс-метода

Классические примеры задач линейного программирования связаны с производством товаров при ограниченных ресурсах (время, рабочая сила или материалы). Чтобы проиллюстрировать, как линейное программирование решает такие задачи, представим фабрику, собирающую два или более продукта, которые должны быть поставлены в определенный срок. Рабочие фабрики производят два продукта, x и y , в течение восьмичасовой смены. Для каждого продукта они получают разную прибыль (которая вычисляется путем вычитания затрат из выручки), разные часовые нормы производства и разный ежедневный спрос на рынке:

- » **выручка в долларах США за каждый продукт:** $x = 15, y = 25$;
- » **производительность в час:** $x = 50, y = 40$;
- » **ежедневный спрос на продукт:** $x = 300, y = 200$.

По сути, бизнес-задача заключается в том, чтобы решить, производить больше продукта x , который легче собирать, но приносит меньше прибыли, или продукта y , который гарантирует больший доход, но меньшее производство. Для решения задачи сначала определим целевую функцию. Выразим ее как сумму количеств двух продуктов, умноженных на их ожидаемую выручку за единицу, которую нужно максимизировать (минимизировать целевую функцию нужно, только если задача касается затрат):

$$f(x, y) = 15 * x + 25 * y$$

В этой задаче есть неравенства, ограниченные значениями x и y , которые должны выполняться для получения правильного результата оптимизации:

$$\begin{aligned} 0 &\leq x \leq 300 \\ 0 &\leq y \leq 200 \end{aligned}$$

Действительно, нельзя произвести отрицательное количество продуктов и нет смысла производить больше продуктов, чем требует рынок. Другое важное ограничение — доступное время, поскольку нельзя превышать 8 часов на каждую рабочую смену. Это означает, что нужно рассчитать время на производство продуктов x и y и ограничить общее время не более чем 8 часами.

$$x / 40 + y / 50 \leq 8$$

Функции можно представить на декартовой плоскости. (Чтобы освежить знания о построении функций, обратитесь к <https://www.mathplanet.com/education/pre-algebra/graphing-and-functions/linear-equations-in-the-coordinate-plane>.) Поскольку в этой задаче все можно выразить с помощью функций, задачи линейного программирования также можно решать как геометрические задачи в декартовом координатном пространстве. Если задача не включает более двух переменных, можно построить график двух переменных и их ограничений в виде линий на плоскости и определить, как они ограничивают геометрическую фигуру. Вы обнаружите, что линии ограничивают область в форме многоугольника, называемую *допустимой областью*. В этой области находится решение, содержащее все допустимые (согласно ограничениям) входные данные для задачи.

Когда задача имеет дело более чем с двумя переменными, ее все еще можно представить с помощью пересекающихся линий в пространстве, но визуально это изобразить нельзя, поскольку каждая входная переменная требует измерения на графике, а графики ограничены тремя измерениями мира, в котором мы живем.

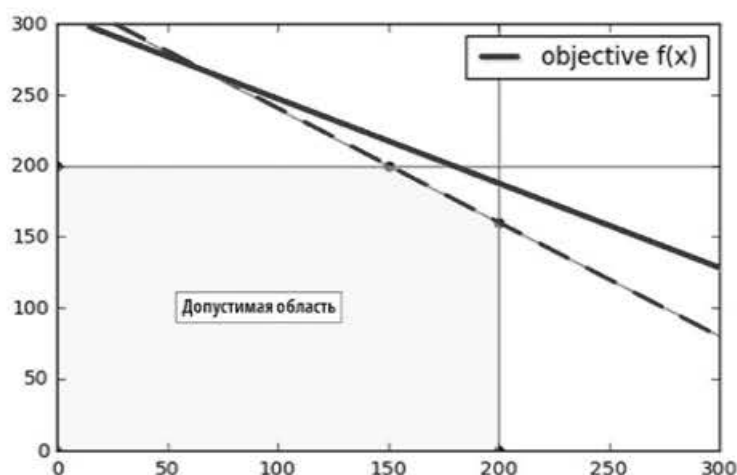
Алгоритм линейного программирования исследует ограниченную допустимую область разумным способом и выдает решение. На самом деле, вам не нужно проверять каждую точку в ограниченной области, чтобы определить наилучшее решение задачи. Представьте целевую функцию как еще одну линию на плоскости (ведь целевая функция тоже линейная). Вы увидите, что искомое решение — это координаты точек, где допустимая область и линия целевой функции впервые касаются друг друга (см. рисунок 19.1). Когда линия целевой функции спускается сверху (приходя извне допустимой области, где находятся результаты, которые нельзя принять из-за ограничений), в определенный момент она коснется области. Эта точка касания обычно является вершиной области, но может быть и целой стороной многоугольника (в этом случае каждая точка на этой стороне является оптимальным решением).

На практике симплекс-алгоритм не может визуально опускать линии, как в этом примере. Вместо этого он движется вдоль границы допустимой области (перебирая вершины) и проверяет значения целевой функции в каждой вершине, пока не найдет решение. Следовательно, фактическое время выполнения зависит от количества вершин, которое, в свою очередь, зависит от количества ограничений и переменных, участвующих в решении. (Больше переменных означает больше измерений и больше вершин.)

Понимание ограничений

По мере того как вы приобретаете больше уверенности в линейном программировании и задачи становятся сложнее, вам требуются более комплексные подходы, чем базовый симплекс-алгоритм, представленный в этой главе. Фактически оригинальный симплекс-метод больше не используется, поскольку его заменили более сложные алгоритмы, которые геометрически прорезают внутреннюю часть допустимой области вместо того, чтобы двигаться вдоль нее. Эти новые алгоритмы используют короткий путь, когда очевидно, что алгоритм ищет решение не с той стороны области.

РИСУНОК 19.1.
Поиск точки,
где целевая
функция
касается
допустимой
области



Работа с числами с плавающей точкой может быть ограничивающей, поскольку многие задачи требуют бинарного (1/0) или целочисленного ответа. (В примере с двумя продуктами нельзя производить дробные количества продуктов. Значения x и y должны быть целыми числами.) Более того, другие задачи могут требовать использования кривых, а не прямых линий для корректного представления пространства задачи и допустимой области. Алгоритмы целочисленного линейного программирования и нелинейного программирования реализованы в коммерческом программном обеспечении. Просто имейте в виду, что и целочисленное, и нелинейное программирование — это NP-полные задачи, которые могут потребовать столько же времени, если не больше, чем другие известные вам алгоритмы.

§ 2. Использование линейного программирования на практике

Лучший способ начать работу с линейным программированием — использовать готовые решения, а не создавать собственные приложения. Первый блок, который следует далее, поможет вам установить готовое решение, используемое в последующих примерах.



ВНИМАНИЕ

При работе с программным продуктом вы можете обнаружить существенные различия между программным обеспечением с открытым исходным кодом и коммерческими пакетами. Хотя программное обеспечение с открытым исходным кодом предлагает широкий спектр алгоритмов, производительность может разочаровывать при работе с большими и сложными задачами. В реализации алгоритмов линейного программирования как части рабочего программного обеспечения все еще много искусства, и не стоит ожидать, что программное обеспечение с открытым исходным кодом будет работать так же быстро и гладко, как коммерческие решения.

Тем не менее открытый исходный код предоставляет несколько хороших вариантов для изучения линейного программирования. В следующих блоках используется решение с открытым исходным кодом для Python под названием PuLP, которое позволяет создавать оптимизации линейного программирования после определения целевой функции и ограничений в виде функций Python. Это в основном дидактическое решение, подходящее для того, чтобы помочь вам проверить, как работает линейное программирование на некоторых задачах, и получить представление о формулировке задач в математических терминах.



ПРИМЕЧАНИЕ

PuLP предоставляет интерфейс к базовым программам-решателям. Python поставляется с решателем по умолчанию с открытым исходным кодом, доступ к которому обеспечивает PuLP. Производительность (скорость, точность и масштабируемость), которую обеспечивает PuLP, почти полностью зависит от решателя и оптимизатора, которые выбирает пользователь. Лучшие решатели — это коммерческие продукты, такие как CPLEX (<https://www.ibm.com/analytics/cplex-optimizer>), XPRESS (<https://www.fico.com/fico-xpress-optimization/docs/latest/overview.html>) и GuRoBi (<https://www.gurobi.com>), которые обеспечивают огромное преимущество в скорости в сравнении с решателями с открытым исходным кодом.

Настройка PuLP дома

PuLP — это проект Python с открытым исходным кодом, созданный Жан-Себастьяном Руа и позже модифицированный и поддерживаемый Стюартом Энтони Митчеллом. Пакет PuLP помогает определять задачи линейного программирования и решать их с помощью встроенного решателя (который использует симплекс-метод). Вы также можете использовать другие решатели, доступные в публичных репозиториях или по платной лицензии. Репозиторий проекта (содержащий весь исходный код и множество примеров) находится по адресу <https://github.com/coin-or/pulp>. Полная документация расположена по адресу <https://coin-or.github.io/pulp>.

PuLP не входит в стандартную поставку дистрибутива Anaconda, поэтому вам нужно установить его самостоятельно. Для установки PuLP нужно использовать командную строку Anaconda3 (или выше), так как более старые версии командной строки Anaconda не подойдут. Откройте командную оболочку, введите: **pip install pulp** и нажмите Enter. При наличии доступа к интернету команда **pip** загрузит пакет PuLP и установит его в Python. (В примерах этой главы используется PuLP версии 1.6.1, но более поздние версии должны обеспечивать ту же функциональность.) Для работы с примером потребуется выполнять операции с векторами значений, поэтому вам также понадобится библиотека NumPy (просто введите: **pip install numpy**, если она отсутствует в вашей среде Python).

Оптимизация производства и выручки

Задача в этом блоке — еще один пример оптимизации, связанной с производством. Вы работаете с двумя продуктами (поскольку это подразумевает всего две переменные, которые можно представить на двумерной диаграмме) — A и B , которые должны пройти серию преобразований через три этапа. Каждый этап требует определенного количества операторов (значение n), которыми могут быть рабочие или роботы, и каждый этап работает максимум определенное количество дней в месяц (представлено значением t). Каждый этап по-разному обрабатывает каждый продукт, требуя разного количества дней до завершения. Например, рабочему на первом этапе (называемом 'res_1') требуется два дня для завершения продукта A , но три дня — для продукта B . Наконец, у каждого продукта разная прибыль: продукт A приносит 3000 \$ за единицу, а продукт B — 2500 \$ за единицу. В следующей таблице представлено краткое описание задачи:

Этап производства	Время на продукт А на рабочего (дней)	Время на продукт В на рабочего (дней)	Время работы (дней)	Количество рабочих
res_1	2	3	30	2
res_2	3	2	30	2
res_3	3	3	22	3

Чтобы найти целевую функцию, вычислите сумму количества каждого продукта, умноженного на его прибыль. Ее нужно максимизировать. Хотя это явно не указано в задаче, существуют определенные ограничения. Во-первых, время работы ограничивает производительность на каждом этапе. Во-вторых — количество рабочих. В-третьих — производительность относительно типа обрабатываемого продукта. Задачу можно переформулировать как сумму времени, используемого для обработки каждого продукта на каждом этапе, которая не может превышать время работы, умноженное на количество доступных рабочих. Количество рабочих, умноженное на количество рабочих дней, дает вам временные ресурсы, которые вы можете использовать. Эти ресурсы не могут быть меньше времени, необходимого для производства всех запланированных к поставке продуктов.

```
objective = 3000 * qty_A + 2500 * qty_B
production_rate_A * qty_A + production_rate_B * qty_B
<= uptime_days * workers
```

Вы можете выразить каждое ограничение, используя количество одного продукта для определения другого (фактически, если вы производите *A*, вы не можете производить *B*, когда производство *A* не оставляет времени).

```
qty_B <= ((uptime_days * workers) -
(production_rate_A * qty_A)) / production_rate_B
```

Для удобства доступа вы можете записать все значения, относящиеся к каждому этапу для `production_rate_A`, `production_rate_B`, `uptime_days` и `workers`, в словарь Python. Прибыль же храните в переменных. (Этот код можно найти в файле `A4D2E: 19: Linear Programming.ipynb`, доступном для скачивания; подробнее см. во введении.)

```
import numpy as np
import matplotlib.pyplot as plt
import pulp
```



```

res_1 = {'A':2, 'B':3, 't':30, 'n':2}
res_2 = {'A':3, 'B':2, 't':30, 'n':2}
res_3 = {'A':3, 'B':3, 't':22, 'n':3}
res = {'res_1': res_1, 'res_2': res_2, 'res_3': res_3}
profit_A = 3000
profit_B = 2500

```

После того как задача оформлена в подходящую структуру данных, попробуйте визуализировать ее с помощью функций построения графиков Python. Установите продукт *A* как абсциссу и, поскольку решение неизвестно, представьте производство продукта *A* как вектор количеств от 0 до 30 (количества не могут быть отрицательными). Что касается продукта *B* (как видно из формулировок выше), выведите его из оставшегося производства после выполнения *A*. Сформулируйте три функции, по одной для каждого этапа, чтобы при определении количества для *A* вы получали соответствующее количество *B* с учетом ограничений.

```

a = np.linspace(0, 30, 30)
c1 = ((res['res_1']['t'] * res['res_1']['n']) -
       res['res_1']['A'] * a) / res['res_1']['B']
c2 = ((res['res_2']['t'] * res['res_2']['n']) -
       res['res_2']['A'] * a) / res['res_2']['B']
c3 = ((res['res_3']['t'] * res['res_3']['n']) -
       res['res_3']['A'] * a) / res['res_3']['B']

plt.figure(figsize=(12, 6))
plt.plot(a, c1, label='constrain #1')
plt.plot(a, c2, label='constrain #2')
plt.plot(a, c3, label='constrain #3')

axes = plt.gca()
axes.set_xlim([0,30])
axes.set_ylim([0,30])
plt.xlabel('qty model A')
plt.ylabel('qty model B')

border = np.array((c1,c2,c3)).min(axis=0)

plt.fill_between(a, border, color='yellow', alpha=0.5)
plt.scatter(*zip(*[(0,0), (20,0),
                   (0,20), (16,6), (6,16)]))

plt.legend()
plt.show()

```

Ограничения превращаются в три линии на графике, как показано на рисунке 19.2. Линии пересекаются между собой, показывая допустимую область. Это область, ограниченная тремя линиями, где значения A и B всегда меньше или равны значениям на любой из линий ограничений. (Ограничения представляют собой границу; вы не можете иметь значения A или B за их пределами.)

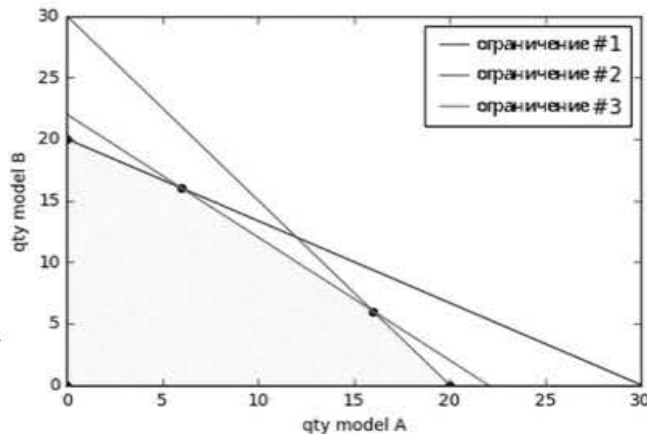


РИСУНОК 19.2.
Определение
правильной
вершины

Согласно симплекс-методу, оптимальное решение — это одна из пяти вершин многоугольника, которыми являются $(0, 0)$, $(20, 0)$, $(0, 20)$, $(16, 6)$ и $(6, 16)$. Чтобы найти решение, нужно настроить необходимые функции из пакета PuLP. Сначала определите задачу и назовите ее `model`. При этом вы указываете, что это задача максимизации и что обе переменные A и B должны быть положительными:

```
model = pulp.LpProblem("max_profit", pulp.LpMaximize)
A = pulp.LpVariable('A', lowBound=0)
B = pulp.LpVariable('B', lowBound=0)
```



СОВЕТ

Решатель PuLP также может искать целочисленные решения, чего не может делать оригинальный симплекс-метод. Просто добавьте параметр `cat='Integer'` при определении переменной: `A = pulp.LpVariable('A', lowBound=0, cat='Integer')` — и получите только целые числа в качестве решения. Имейте в виду, что в некоторых задачах целочисленные результаты могут оказаться менее оптимальными, чем результаты с десятичными числами, поэтому используйте целочисленное решение, только если это имеет смысл для вашей задачи (например, вы не можете произвести дробную часть продукта).

Далее добавьте целевую функцию, просуммировав две переменные, определенные через `pulp.LpVariable` и представляющие идеальные количества продуктов *A* и *B*, умноженные на значение прибыли за единицу каждого:

```
model += profit_A * A + profit_B * B
```

Наконец, добавьте ограничения точно таким же образом, как и целевую функцию. Создайте формулировку, используя соответствующие значения (взятые из словаря данных) и предопределенные переменные *A* и *B*:

```
model += res['res_1']['A'] * A + res['res_1']['B'] * B <= res['res_1']['t'] * res['res_1']['n']
model += res['res_2']['A'] * A + res['res_2']['B'] * B <= res['res_2']['t'] * res['res_2']['n']
model += res['res_3']['A'] * A + res['res_3']['B'] * B <= res['res_3']['t'] * res['res_3']['n']
```

Модель готова к оптимизации (она включает как целевую функцию, так и ограничения). Вызовите метод `solve`, а затем проверьте его статус. (Иногда решение может оказаться невозможно найти или оно может быть неоптимальным.)

```
model.solve()
print(f"Completion status: {pulp.LpStatus[model.status]}")
Completion status: Optimal
```

Получив подтверждение, что оптимизатор нашел оптимальное решение, выведите соответствующие количества продуктов *A* и *B*:

```
print(f"Production of model A = {A.varValue:0.1f}")
print(f"Production of model B = {B.varValue:0.1f}")

Production of model A = 16.0
Production of model B = 6.0
```

Кроме того, выведите итоговую прибыль, достижимую при таком решении:

```
print(f"Maximum profit achieved: {pulp.value(model.
        objective):0.1f}")

Maximum profit achieved: 63000.0
```


В ЭТОЙ ГЛАВЕ

- » Понимание, когда эвристики полезны для алгоритмов
- » Изучение, почему поиск пути может быть сложным для робота
- » Быстрый старт с использованием поиска по первому наилучшему совпадению (best-first search)
- » Улучшение алгоритма Дейкстры и выбор наилучшего эвристического маршрута с помощью алгоритма A^*

ГЛАВА 20

Рассмотрение эвристических методов

В качестве заключительной темы об алгоритмах эта глава завершает обзор эвристических методов, начатый в главе 18, где эвристика описывается как эффективный способ использования локального поиска для навигации по соседним решениям. В главе 18 эвристические методы определяются как обоснованные предположения о решении, то есть это наборы эмпирических правил, указывающих на желаемый результат и помогающих алгоритмам делать к нему правильные шаги; однако сами по себе эвристические методы не могут точно указать, как достичь решения.

Глава начинается с рассмотрения разновидностей эвристических методов. Некоторые из них сейчас весьма популярны благодаря применению в ИИ и робототехнике: роевые эвристики, метаэвристики, моделирование на основе машинного обучения и эвристическая маршрутизация. Далее в главе обсуждается, как эвристические методы используются мобильными роботами для безопасной и эффективной навигации по известным и неизвестным территориям ради выполнения своих задач. Все такие эвристики основаны на расстоянии, и в блоке «Использование мер расстояния в качестве эвристики» представлено, как вычисляются

и могут помочь в различных проблемных ситуациях две наиболее распространенные меры расстояния — евклидово расстояние и манхэттенское расстояние.

Глава завершается демонстрацией двух мощных алгоритмов поиска пути, помогающих роботам безопасно достигать своих целей, — алгоритма поиска по первому наилучшему совпадению и алгоритма A^* (произносится как «А-звездочка»). Алгоритм A^* получил известность благодаря роботу по имени Шейки, как объясняется в главе. Прежде чем показать код на Python, демонстрирующий работу этих алгоритмов, в главе также рассматривается, как эффективно создать лабиринт (что само по себе алгоритм), чтобы смоделировать среду с определенными сложностями (препятствиями), по которой эвристические алгоритмы поиска пути должны проложить маршрут.



ЗАПОМНИТЕ

Вам не нужно вручную набирать исходный код для этой главы. На самом деле, гораздо проще использовать загружаемые исходники. Исходный код главы можно найти в файле `A4D2E; 20; Heuristic Algorithms.ipynb` из загружаемых материалов. Подробнее о том, как найти этот файл, смотрите во введении.

§ 1. Дифференциальная эвристика

Слово *эвристика* происходит от древнегреческого *heuriskein*, что означало «изобретать» или «открывать». Его первоначальное значение подчеркивает тот факт, что применение эвристики — это практический способ найти решение, которое четко не определено, но обнаруживается путем исследования и интуитивного понимания общего направления действий. Эвристики полагаются на удачные догадки или метод проб и ошибок, пробуя различные решения. *Эвристический алгоритм*, то есть алгоритм, основанный на эвристиках, решает задачу быстрее и эффективнее с точки зрения вычислительных ресурсов, жертвуя при этом точностью и полнотой решения. Эти решения контрастируют с большинством алгоритмов, которые рассматривались в книге, и обладают определенными гарантиями результата. Когда задача становится слишком сложной, эвристический алгоритм может оказаться единственным способом получить решение.

Нужно понимать, что у эвристик, как и у истины, есть свои оттенки. Эвристики затрагивают передовые рубежи разработки алгоритмов сегодня. Революция в области искусственного интеллекта основывается на представленных ранее в книге алгоритмах, которые упорядочивают, организуют, ищут и обрабатывают входные данные. На вершине ис-

рархии находятся эвристические алгоритмы, которые обеспечивают оптимизацию, а также поиск, определяющий, как машины учатся на данных и становятся способными решать задачи автономно, без прямого вмешательства.

Эвристики не волшебная палочка; нет решения, которое подходило бы для всех задач. У эвристических алгоритмов есть серьезные недостатки, и нужно знать, когда их использовать. Кроме того, эвристики могут приводить к неверным выводам как для компьютеров, так и для людей.

Фактически для людей повседневные эвристики, экономящие время, часто оказываются ошибочными, например, когда мы предвзято оцениваем человека или ситуацию с первого взгляда. Даже более обоснованные правила поведения, базирующиеся на опыте, приводят к правильному решению только при определенных обстоятельствах. Например, рассмотрим привычку стучать по электроприборам, когда они не работают. Если проблема в плохом контакте, удар по прибору может помочь, восстановив электрическое соединение; но нельзя делать битье по приборам общей эвристикой, потому что в других случаях такое «решение» может оказаться неэффективным или даже серьезно повредить прибор.

Рассматривая цели эвристик

Эвристики могут ускорить длительный полный перебор, выполняемый другими решениями, особенно в случае NP-трудных задач, требующих экспоненциального числа попыток в зависимости от количества входных данных. Например, рассмотрим задачу коммивояжера или варианты задачи выполнимости булевых формул (2-выполнимость рассматривается в главе 18), такие как MAX-3SAT (дополнительные подробности см. на <https://people.maths.bris.ac.uk/~csxam/bccs/approx.pdf>). Эвристики определяют направление поиска с помощью оценки, что позволяет исключить большое количество комбинаций, которые пришлось бы проверять в противном случае.

Поскольку эвристика — это оценка или предположение, она может привести использующий ее алгоритм к неверному заключению, которое может быть неточным или просто субоптимальным. *Субоптимальное решение* — это решение, которое работает, но не является наилучшим из возможных. Например, при численной оценке эвристика может дать решение, отличающееся от правильного ответа на несколько чисел. Такое субоптимальное решение может быть приемлемым во многих ситуациях, но некоторые ситуации требуют точного решения. Другие проблемы, часто связанные с эвристиками, — это невозможность найти все лучшие решения, а также непостоянство времени и вычислений,

необходимых для достижения решения. Эвристика идеально подходит при работе с алгоритмами, которые в противном случае потребовали бы больших затрат при взаимодействии с другими алгоритмическими методами. Например, некоторые задачи невозможно решить без эвристик из-за низкого качества и огромного количества входных данных. Задача коммивояжера (TSP) — одна из таких: если нужно объехать большое количество городов, нельзя использовать точные методы. TSP и другие подобные задачи исключают любое точное решение. Приложения искусственного интеллекта попадают в эту категорию, поскольку многие задачи ИИ, такие как распознавание устной речи или содержимого изображения, не решаются точной последовательностью шагов и правил.

От генетики к ИИ

В обсуждении локального поиска в главе 18 представлены такие эвристики, как имитация отжига и поиск с запретами. Эти эвристики используются для помощи в оптимизации восхождением к вершине (оптимизация, которая помогает не застревать на неоптимальных решениях). Помимо этих, семейство эвристик включает много различных приложений, среди которых:

- » **роевой интеллект** — набор эвристик, основанных на изучении поведения роев насекомых (таких, как пчелы, муравьи или светлячки) или частиц. Метод использует множественные попытки найти решение с помощью агентов, которые кооперативно взаимодействуют друг с другом и условиями задачи. *Агент* — это независимая вычислительная единица, например, когда запускается несколько независимых экземпляров одного и того же алгоритма, каждый из которых работает над своей частью задачи. Профессор Марко Дориго, один из ведущих экспертов и исследователей в области алгоритмов роевого интеллекта, предоставляет дополнительную информацию по этой теме на <https://scholar.google.com/citations?user=PwYT6EMAAAAJ> (где содержится список его различных работ);
- » **метаэвристика** — эвристические методы, которые помогают определить (или даже создать) правильную эвристику для вашей задачи. Среди метаэвристических методов наиболее известны *генетические алгоритмы*, вдохновленные естественной эволюцией. Генетические алгоритмы начинают работу с набора возможных решений проблемы, а затем генерируют новые решения, используя мутацию (добавляют или удаляют что-то в решении) и кроссовер (смешивают части различных решений, когда решение можно разделить). Например, в задаче о расстановке n

ферзей (глава 18) вы увидите, что шахматную доску можно разделить вертикально на части, поскольку ферзи не перемещаются горизонтально в рамках решения задачи, что делает ее подходящей для применения кроссовера. Когда набор достаточно велик, генетические алгоритмы отбирают выживающие решения, отбрасывая те, которые не работают или не имеют перспектив. Затем отобранный набор проходит через очередную итерацию мутации, кроссовера и отбора. После достаточного количества времени и итераций генетические алгоритмы могут найти решения, которые работают лучше и полностью отличаются от исходных;

- » **машинное обучение.** Такие подходы, как нейронечеткие системы, машины опорных векторов и нейронные сети, являются основой того, как компьютер учится оценивать и классифицировать на основе обучающих примеров, предоставляемых в составе наборов данных. Подобно тому, как ребенок учится на опыте, алгоритмы машинного обучения определяют, как выдать наиболее правдоподобный ответ без использования точных правил и детальных инструкций (подробнее о том, как работает машинное обучение, читайте в книге «*Machine Learning For Dummies*», 2-е издание, авторы Джон Пол Мюллер и Лука Массарон [издательство Wiley]);
- » **эвристическая маршрутизация** – набор эвристик, которые помогают роботам (но также используются в сетевой телекоммуникации и транспортной логистике) выбирать оптимальный путь и избегать препятствий при передвижении.

§ 2. Маршрутизация роботов с использованием эвристики

Управление роботом в неизвестной среде означает обход препятствий и поиск пути к определенной цели. Это одновременно фундаментальная и сложная задача в искусственном интеллекте. Роботы могут полагаться на различные датчики, такие как *лазерный дальномер*, *лидар* (устройства, позволяющие определять расстояние до объекта с помощью лазерного луча) или *гидролокационные массивы* (устройства, использующие звук, чтобы «видеть» окружающую среду) для навигации в окружающем пространстве. Тем не менее, какой бы сложной ни была аппаратура, которой оснащены роботы, им все равно нужны правильные алгоритмы:

- » для поиска кратчайшего пути к месту назначения (или хотя бы достаточно короткого);
- » обхода препятствий на пути;
- » выполнения специальных действий, таких как минимизация поворотов или торможение.

Алгоритм *поиска пути* (также называемый *планированием маршрута*, или просто *прокладкой пути*) помогает роботу начать движение из одной точки и достичь другой, используя кратчайший путь между ними, предвидя и избегая препятствия на пути. (Недостаточно просто реагировать после столкновения со стеной.) Поиск пути также полезен при перемещении любого другого устройства к цели в пространстве, даже виртуальном, например в видеоигре или на веб-страницах.



ЗАПОМНИТЕ

Автономная маршрутизация является ключевой возможностью беспилотных автомобилей (SDC) — транспортных средств, которые могут воспринимать дорожную обстановку и двигаться к месту назначения без вмешательства человека. (Вам все равно нужно сказать машине, куда ехать: она не умеет читать мысли.) Эта недавняя статья из журнала *Automotive World* предлагает хороший обзор разработок и ожиданий относительно беспилотных автомобилей: <https://www.automotiveworld.com/articles/the-future-of-driving-what-to-expect-in-2021>.

Разведка на неизвестной территории

Алгоритмы поиска пути выполняют все ранее обсуждавшиеся задачи для нахождения кратчайшего маршрута, обхода препятствий и реализации других желаемых действий. Алгоритмы работают, используя базовые схематические карты окружающего пространства. Эти карты бывают двух видов:

- » **топологические карты** – упрощенные схемы, из которых удалены все несущественные детали. Такие карты сохраняют ключевые ориентиры, правильные направления и некоторые пропорции масштаба для расстояний. Реальные примеры топологических карт включают схемы метро Токио (<https://www.tokymetro.jp/en/subwaymap>) и Лондона (<https://tfl.gov.uk/maps/track/tube>);
- » **карты занятости**. Эти карты разделяют окружающее пространство на маленькие пустые квадраты или шестиугольники, заполняя их, когда датчики робота обнаруживают препятствие в соответствующей точке пространства. Пример такой карты можно увидеть в Чешском техническом университете в Праге (<http://cmp.felk.cvut.cz/cmp/demos/Omni/mobil>). Кроме того, посмотрите видеоролики, показывающие, как робот строит и визуализирует такую карту (<https://www.youtube.com/watch?v=zjl7NmutMIc> и <https://www.youtube.com/watch?v=RhPlzIyTT58>).

И топологические карты, и карты занятости можно визуализировать как графические диаграммы. Однако для алгоритмов они лучше воспринимаются, когда преобразованы в подходящую структуру данных. Лучшая структура данных для этой цели — граф, поскольку вершины могут легко представлять квадраты, шестиугольники, ориентиры и путевые точки. Ребра могут соединять вершины так же, как это делают дороги, проходы и пути.



ЗАПОМНИТЕ

Ваш GPS-навигатор работает с использованием графов. За непрерывной, детальной, красочной картой, которую устройство отображает на экране, скрываются дорожные карты, представленные в виде наборов вершин и ребер, по которым перемещаются алгоритмы, помогающие вам найти путь и избежать пробок.

Представление территории робота в виде графа возвращает нас к проблемам, обсуждавшимся в главе 9, где рассматривалось, как перемещаться от одной вершины к другой по кратчайшему пути. Кратчайший путь может быть путем, проходящим через наименьшее количество вершин, или

путем, имеющим наименьшую стоимость (учитывая сумму весов пересекаемых ребер, которые могут представлять длину ребра или какую-либо другую стоимость). Как и при вождении автомобиля, вы выбираете маршрут не только на основе расстояния до пункта назначения, но и с учетом трафика (загруженных дорог или заблокированных пробками), состояния дорог и ограничений скорости, которые могут повлиять на качество вашего путешествия.

При поиске кратчайшего пути к пункту назначения в графе простейшими и базовыми алгоритмами в теории графов являются поиск в глубину и алгоритм Дейкстры (описанный в главе 9). Поиск в глубину исследует граф, продвигаясь как можно дальше от начальной точки, а затем возвращается для изучения других путей, пока не найдет пункт назначения. Алгоритм Дейкстры исследует граф разумным и жадным способом, рассматривая только кратчайшие пути. Несмотря на свою простоту, оба алгоритма чрезвычайно эффективны при анализе простого графа, как при обзоре с высоты птичьего полета, когда есть полное представление о направлениях, которые нужно выбрать для достижения цели, и низкие затраты на оценку различных возможных путей.

Ситуация с роботом несколько иная, поскольку он не может воспринимать все возможные пути одновременно из-за ограниченной видимости и дальности обзора (препятствия могут скрывать путь, или цель может находиться слишком далеко). Робот исследует окружающую среду по мере движения и в лучшем случае может оценить расстояние и направление до конечного пункта назначения. Это похоже на решение лабиринта, но не как в головоломке-лабиринте, а, скорее, как погружение в живой лабиринт из кустарника, где вы можете чувствовать направление своего движения или заметить пункт назначения вдалеке.



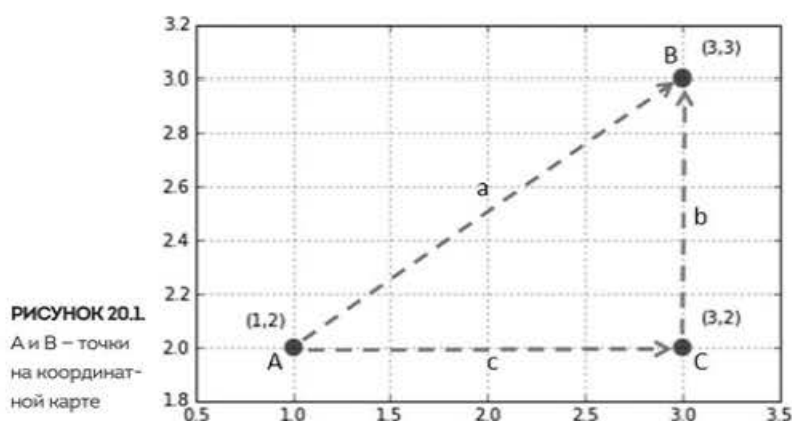
СОВЕТ

Живые лабиринты встречаются по всему миру. Некоторые из самых известных были построены в Европе с середины XVI до XVIII века. В живом лабиринте вы не видите, куда идете, потому что живые изгороди слишком высокие. Вы можете определять направление, если видите солнце, и даже заметить цель (см. пример на <https://www.venetoinside.com/hidden-treasures/post/maze-of-villa-pisani-in-stra-venice>). Известные живые лабиринты также можно увидеть в таких фильмах, как «Сияние» Стэнли Кубрика и «Гарри Поттер и Кубок огня».

Использование мер расстояния в качестве эвристики

Когда невозможно решить реальные задачи точным алгоритмическим способом из-за того, что входные данные запутаны, неполны или нестабильны, может помочь использование эвристики. При поиске пути с использованием координат на декартовой плоскости (плоских картах, основанных на наборе горизонтальных и вертикальных координат) две простые меры могут определять расстояния между двумя точками на этой плоскости — евклидово расстояние и манхэттенское расстояние.

Люди часто используют евклидово расстояние, поскольку оно основано на теореме Пифагора о треугольниках и представляет собой кратчайшее расстояние между двумя точками. Если хотите узнать расстояние по прямой видимости между двумя точками на плоскости, скажем A и B , и знаете их координаты, можно представить их как концы гипотенузы (самой длинной стороны прямоугольного треугольника). Как показано на рисунке 20.1, расстояние вычисляется на основе длины двух других сторон путем создания третьей точки C , горизонтальная координата которой берется от точки B , а вертикальная — от точки A .



Этот процесс сводится к тому, что нужно взять разность между горизонтальными и вертикальными координатами двух точек, возвести обе разности в квадрат (чтобы они стали положительными), сложить их и, наконец, извлечь квадратный корень из результата. В данном примере для перемещения из A в B используются координаты (1, 2) и (3, 3):

$$\text{sqrt}((1-3)^2 + (2-3)^2) = \text{sqrt}(2^2+1^2) = \text{sqrt}(5) = 2.236$$



ЗАПОМНИТЕ

На самом деле, измерить расстояния на поверхности Земли точно с помощью евклидова расстояния невозможно, поскольку ее поверхность не плоская, а близка к сферической. По мере увеличения расстояния между измеряемыми точками на поверхности Земли погрешность евклидова расстояния тоже будет возрастать. Для больших расстояний более подходящими являются расстояние по формуле гаверсинуса, основанное на координатах широты и долготы (см. на <https://www.igismap.com/haversine-formula-calculate-geographic-distance-earth>), или более точное расстояние Винсенти (см. на <https://metacpan.org/pod/GIS::Distance::Vincenty>).

Манхэттенское расстояние работает иначе. Вы начинаете с суммирования длин сторон b и c , что равносильно суммированию абсолютных значений разностей между горизонтальными и вертикальными координатами точек A и B .

$$|(1-3)| + |(2-3)| = 2 + 1 = 3$$

Евклидово расстояние отмечает кратчайший путь, а манхэттенское расстояние дает самый длинный, но наиболее правдоподобный маршрут, если вы ожидаете препятствия при движении по прямому пути. Фактически такое движение представляет траекторию такси на Манхэттене (отсюда и название), перемещающегося вдоль городского квартала к месту назначения (короткий путь через здания никогда не сработает). Другие названия этого подхода — расстояние городских кварталов или расстояние такси. Следовательно, если вам нужно переместиться из A в B , но вы не знаете, встретятся ли препятствия между ними, обход через точку C — хорошая эвристика, поскольку это расстояние, которое вы ожидаете в худшем случае.

§ 3. Объяснение алгоритмов поиска пути

Последний параграф главы посвящен объяснению двух алгоритмов — алгоритма поиска по первому наилучшему совпадению и алгоритма A^* (читается как «А-звездочка»), которые основаны на эвристических методах. В следующих блоках показано, что оба этих алгоритма обеспечивают быстрое решение задачи о лабиринте, представленном в виде топологической карты или карты занятости в форме графа. Оба алгоритма широко используются в робототехнике и разработке видеоигр.

Создание лабиринта

Топологическая карта или карта занятости напоминает лабиринт из живой изгороди, как упоминалось ранее, особенно если между начальной и конечной точками маршрута есть препятствия.

Существуют специализированные алгоритмы как для создания, так и для прохождения лабиринтов, наиболее известные из которых — алгоритм следования вдоль стены (известный с древности: нужно приложить руку к стене и не отрывать ее, пока не выйдешь из лабиринта) и алгоритм Pledge. (Подробнее о семи классификациях лабиринтов можно прочитать на <http://www.astrolog.org/labyrnth/algrithm.htm>). Однако поиск пути принципиально отличается от решения лабиринта, поскольку при поиске пути вы знаете, где должна находиться цель, тогда как алгоритмы решения лабиринта пытаются решить задачу в полном неведении о местонахождении выхода.

Следовательно, процедура моделирования лабиринта с препятствиями, через который должен пройти робот, использует другой, более простой подход. Вместо создания головоломки из препятствий вы создаете граф вершин, расположенных в виде сетки (напоминающей карту), и случайным образом удаляете связи для имитации наличия препятствий. Граф является неориентированным (каждое ребро можно пройти в обоих направлениях) и взвешенным, поскольку перемещение от одной вершины к другой занимает определенное время. В частности, движение по диагонали занимает больше времени, чем движение вверх/вниз или влево/вправо.

Первый шаг — импорт необходимых пакетов Python. Далее код определяет функции евклидова расстояния и манхэттенского расстояния. (Этот код можно найти в загружаемом файле исходного кода **A4D2E; 20; Heuristic Algorithms.ipynb**; подробнее см. во введении.)

```
import string
import networkx as nx
import matplotlib.pyplot as plt
import random

def euclidean_dist(a, b, coord):
    (x1, y1) = coord[a]
    (x2, y2) = coord[b]
    return ((x1 - x2)**2 + (y1 - y2)**2)**0.5

def manhattan_dist(a, b, coord):
    (x1, y1) = coord[a]
    (x2, y2) = coord[b]
```

```

    return abs(x1 - x2) + abs(y1 - y2)

def non_informative(a,b):
    return 0

def ravel(listsoflists):
    return [item for elem in listsoflists for item in elem]

```

Следующий шаг — создание функции для генерации случайных лабиринтов. Она основана на целочисленном начальном значении по вашему выбору, которое позволяет воссоздавать один и тот же лабиринт каждый раз, когда вы указываете то же число. В противном случае генерация лабиринта происходит полностью случайным образом. Нужно также создать общую функцию `node_neighbors()`, поскольку она определяет направление движения из вершины в графе и делает код создания лабиринта более читаемым.

```

def node_neighbors(graph, node):
    return [exit for _,
            exit in graph.edges(node) if exit!=node]

def create_maze(seed=2, drawing=True):
    random.seed(seed)
    letters = [l for l in string.ascii_uppercase[:25]]
    checkboard = [letters[i:i+5]
                  for i in range(0, len(letters), 5)]
    Graph = nx.Graph()
    for j, node in enumerate(letters):
        Graph.add_nodes_from(node)
        x, y = j // 5, j % 5
        x_min = max(0, x-1)
        x_max = min(4, x+1)+1
        y_min = max(0, y-1)
        y_max = min(4, y+1)+1
        adjacent_nodes = ravel(
            [row[y_min:y_max]
             for row in checkboard[x_min:x_max]])
        exits = random.sample(adjacent_nodes,
                              k=random.randint(1, 4))
        for exit in exits:
            if exit not in node_neighbors(Graph, node):
                Graph.add_edge(node, exit)
    spacing = [0.0, 0.2, 0.4, 0.6, 0.8]
    coordinates = [[x, y] for x in spacing \
                   for y in spacing]
    position = {l:c for l,c in zip(letters, coordinates)}

```



```

for node in Graph.nodes:
    for exit in node_neighbors(Graph, node):
        length = int(round(
            euclidean_dist(
                node, exit, position)*10,0))
        Graph.add_edge(node,exit,weight=length)

if drawing:
    draw_params = {'with_labels':True,
                    'node_color':'skyblue',
                    'node_size':700, 'width':2,
                    'font_size':14}
    nx.draw(Graph, position, **draw_params)
    labels = nx.get_edge_attributes(Graph,'weight')
    nx.draw_networkx_edge_labels(Graph, position,
                                edge_labels=labels)

    plt.show()

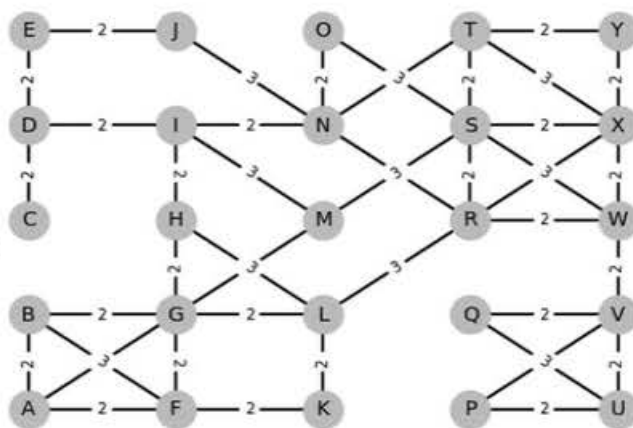
return Graph, position

```

Функции возвращают граф NetworkX (**Graph**) — предпочтительную структуру данных для представления графов, которая содержит 25 вершин (или узлов, если вам так больше нравится) и декартову карту точек (**position**). Вершины размещены на сетке 5×5 , как показано на рисунке 20.2. Результат также применяет функции расстояния и вычисляет положение вершин.

```
graph, coordinates = create_maze(seed=7)
```

РИСУНОК 20.2.
Лабиринт,
представляю-
щий топологи-
ческую карту
с препятстви-
ями





СОВЕТ

В лабиринте, сгенерированном со значением начального числа 7, каждая вершина соединена с остальными. Поскольку процесс генерации случайный, некоторые карты могут содержать несвязанные вершины, что делает невозможным перемещение между ними. Чтобы увидеть, как это работает, попробуйте значение начального числа 6. Такое случается и в реальности; например, иногда робот не может достичь определенного места назначения, потому что оно окружено препятствиями.

В поисках быстрого маршрута по первому наилучшему совпадению

Алгоритм поиска в глубину исследует граф, перемещаясь от вершины к вершине и добавляя направления в структуру данных стека. Когда приходит время двигаться, алгоритм выбирает первое найденное в стеке направление. Это похоже на прохождение лабиринта комнат, где вы выбираете первый увиденный выход. Скорее всего, вы попадете в тупик, который не является вашей целью. Затем приходится возвращаться в ранее посещенные комнаты в поисках другого выхода, что занимает много времени, когда вы находитесь далеко от цели.

Эвристические методы могут существенно помочь с повторениями, создаваемыми стратегией поиска в глубину. Использование эвристики позволяет определить, приближаетесь вы к цели или удаляетесь от нее. Такая комбинация называется алгоритмом *поиска по первому наилучшему совпадению* (BFS). В данном случае слово *наилучший* в названии указывает на то, что при исследовании графа вы не берете первое попавшееся ребро. Вместо этого вы оцениваете, какое ребро выбрать, основываясь на том, какое из них должно привести вас ближе к желаемому результату согласно эвристике. Такое поведение напоминает жадную оптимизацию (выбор лучшего первым), и некоторые также называют этот алгоритм *жадным поиском по первому наилучшему совпадению*. BFS, вероятно, сначала промахнется мимо цели, но благодаря эвристике он не уйдет от нее далеко и будет возвращаться меньше, чем при использовании одного только поиска в глубину.



ЗАПОМНИТЕ

Алгоритм BFS в основном используется в веб-краулерах, которые ищут определенную информацию в интернете. Фактически BFS позволяет программному агенту перемещаться в основном неизвестном графе, используя эвристику для определения того, насколько содержание следующей страницы похоже на исходную (чтобы найти более подходящий контент). Этот алгоритм также широко используется в видеоиграх, помогая управляемым компьютером персонажам перемещаться в поисках врагов и наград, тем самым имитируя жадное, целенаправленное поведение.

Демонстрация реализации поиска по первому наилучшему совпадению (BFS) на Python с использованием ранее построенного лабиринта показывает, как робот может перемещаться в пространстве, рассматривая его как граф. Следующий код содержит общие функции, которые также используются для следующего алгоритма в этом блоке. Функция `graph_weight()` определяет стоимость перехода от одной вершины к другой. Вес представляет расстояние или время.

```
def graph_weight(graph, a, b):  
    return graph.edges[(a, b)][‘weight’]
```

Алгоритм планирования пути моделирует движение робота по графу. Когда он находит решение, план преобразуется в конкретные перемещения. Таким образом, алгоритмы планирования пути выдают результат, указывающий оптимальный способ перемещения от одной вершины к другой; при этом все еще требуется функция для интерпретации этой информации, определения маршрута и расчета длительности пути. Функции `reconstruct_path()` и `compute_path()` предоставляют план в виде последовательности шагов и ожидаемой стоимости на основе результата работы алгоритма планирования пути.

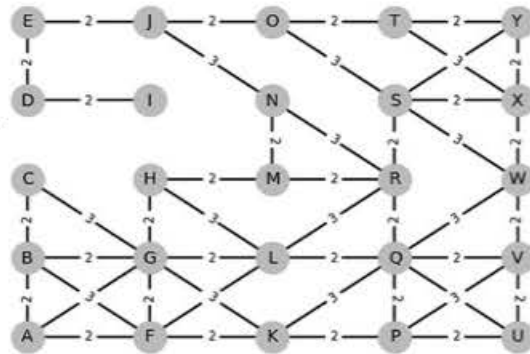
```
def reconstruct_path(connections, start, goal):  
    if goal in connections:  
        current = goal  
        path = [current]  
        while current != start:  
            current = connections[current]  
            path.append(current)  
        return path[::-1]  
  
def compute_path_dist(path, graph):  
    if path:  
        run = 0  
        for step in range(len(path)-1):  
            A = path[step]  
            B = path[step+1]  
            run += graph_weight(graph, A, B)  
        return run  
    else:  
        return 0
```

После подготовки всех базовых функций в примере создается лабиринт с использованием начального значения 30. В этом лабиринте есть несколько основных маршрутов от вершины А до вершины Y, поскольку в центре карты есть препятствия (как показано на рисунке 20.3). На пути также есть тупик (вершина I).


```
graph, coordinates = create_maze(seed=30)
start = 'A'
goal = 'Y'
scoring = manhattan_dist
```

РИСУНОК 20.3.

Сложный лабиринт, который нужно решить с помощью эвристических методов



Реализация BFS немного сложнее, чем код поиска в глубину из главы 9. Она использует два списка: один — для хранения непосещенных вершин (называемый `open_list`), другой — для хранения посещенных вершин (`closed_list`). Список `open_list` действует как очередь с приоритетом, в которой приоритет определяет, какой элемент будет извлечен первым. В данном случае приоритет задается эвристической функцией, таким образом очередь с приоритетом обеспечивает движение в направлении, более близком к цели. Манхэттенское расстояние работает лучше всего в качестве эвристики из-за препятствий, преграждающих путь к месту назначения:

```
# Best-first search
path = {}
open_list = set(graph.nodes())
closed_list = {start: manhattan_dist(start, goal,
                                     coordinates)}

while open_list:

    candidates = open_list & closed_list.keys()
    if len(candidates) == 0:
        print ("Cannot find a way to the goal %s" % goal)
        break
    frontier = [(closed_list[node],
                  node) for node in candidates]
    score, min_node = sorted(frontier)[0]

    if min_node == goal:
        print ("Arrived at final vertex %s" % goal)
```

```

        print ('Unvisited vertices: %i' % (len(
            open_list)-1))
        break
    else:
        print("Processing vertex %s, " % min_node, end="")

    open_list = open_list.difference(min_node)
    neighbors = node_neighbors(graph, min_node)
    to_be_visited = list(neighbors-closed_list.keys())

    if len(to_be_visited) == 0:
        print ("found no exit, retracing to %s"
            % path[min_node])
    else:
        print ("discovered %s" % str(to_be_visited))

    for node in neighbors:
        if node not in closed_list:
            closed_list[node] = scoring(node, goal,
                coordinates)
            path[node] = min_node

    print ('\nBest path is:', reconstruct_path(
        path, start, goal))
    print ('Length of path: %i' % compute_path_dist(
        reconstruct_path(path, start, goal), graph))

```

Подробный вывод примера показывает, как работает алгоритм:

```

Processing vertex A, discovered ['F', 'B', 'G']
Processing vertex G, discovered ['H', 'C', 'L', 'K']
Processing vertex H, discovered ['M']
Processing vertex M, discovered ['R', 'N']
Processing vertex N, discovered ['J']
Processing vertex J, discovered ['E', 'O']
Processing vertex O, discovered ['S', 'T']
Processing vertex T, discovered ['Y', 'X']
Arrived at final vertex Y
Unvisited vertices: 16

Best path is: ['A', 'G', 'H', 'M', 'N', 'J', 'O', 'T', 'Y']
Length of path: 18

```

BFS продолжает движение, пока не закончатся вершины для исследования. Когда все вершины исчерпаны без достижения цели, код сообщает, что невозможно достичь цели и робот не сдвинется с места. Когда код

находит пункт назначения, он прекращает обработку вершин, даже если в `open_list` все еще остаются вершины, что экономит время.

Обнаружение тупика, например попадание в вершину I , означает необходимость поиска ранее неиспользованного маршрута. Благодаря очереди с приоритетом лучшая альтернатива немедленно появляется, и алгоритм выбирает ее. В этом примере BFS эффективно игнорирует 16 вершин и выбирает верхний маршрут на карте, завершая путь от A до Y за 18 шагов.



СОВЕТ

Вы можете протестировать другие лабиринты, задавая разные числа для генератора случайных чисел и сравнивая результаты поиска в ширину с алгоритмом A^* , который будет описан в следующем блоке. Вы обнаружите, что иногда поиск по первому наилучшему совпадению оказывается и быстрым, и точным в выборе оптимального пути, а иногда — нет. Если вам нужен робот, который ищет путь быстро, поиск по первому наилучшему совпадению — лучший выбор.

Эвристический обход с помощью A^*

Алгоритм A^* быстро находит кратчайшие пути в графе, комбинируя жадный поиск Дейкстры, рассмотренный в главе 9, с ранней остановкой (алгоритм прекращает работу при достижении целевой вершины) и эвристической оценкой (обычно основанной на манхэттенском расстоянии), которая подсказывает, какую область графа исследовать в первую очередь. A^* был разработан в Центре искусственного интеллекта Стэнфордского исследовательского института (ныне известного как SRI International) в 1968 году в рамках проекта робота по имени Шейки. Шейки стал первым мобильным роботом, способным самостоятельно решать, как добраться до нужного места, хотя его возможности ограничивались перемещением между несколькими комнатами в лаборатории. (Если посмотрите видео по ссылке <https://www.youtube.com/watch?v=qXdn6ynwpiI>, то увидите, что трясется не робот, а человек, снимающий видео. Но, как всегда, проще обвинить робота.) Шейки стал важной вехой в робототехнике, продемонстрировав, что уже в конце 1960-х годов было технологически возможно создать мобильного робота, работающего без присмотра человека.

Чтобы сделать Шейки полностью автономным, его разработчики создали алгоритм A^* , преобразование Хафа (метод обработки изображений для обнаружения границ объектов) и метод графа видимости (способ представления пути в виде графа). Статья по адресу <http://www.ai.sri.com/shakey> подробнее рассказывает о Шейки и даже показывает его в действии. В настоящее время алгоритм A^* считается лучшим доступным

алгоритмом для поиска кратчайшего пути в графе, когда приходится иметь дело с неполной информацией и предположениями (которые учитываются эвристической функцией, направляющей поиск). A^* способен:

» **каждый раз находить решение с кратчайшим путем.**

Алгоритм может сделать это, если такой путь существует и если A^* правильно информирован эвристической оценкой. A^* работает на основе алгоритма Дейкстры, который гарантированно всегда находит оптимальное решение;

» **находить решение быстрее любого другого алгоритма.**

A^* может делать это при наличии адекватной эвристики – той, которая указывает правильное направление для достижения цели схожим способом, хотя и более умным, чем поиск по первому наилучшему совпадению;

» **вычисляет веса при проходе по ребрам.** Веса учитывают стоимость перемещения в определенном направлении. Например, поворот может занять больше времени, чем движение по прямой, как в случае с роботом Шейки.



ЗАПОМНИТЕ

Правильная, справедливая и *допустимая* эвристика предоставляет алгоритму A^* полезную информацию о расстоянии до цели, никогда не переоценивая стоимость ее достижения. Более того, A^* использует свою эвристику интенсивнее, чем поиск в ширину, поэтому эвристическая функция должна выполнять вычисления быстро, иначе общее время обработки будет слишком большим.

В этом примере реализация на Python использует тот же код и структуры данных, что и поиск в ширину, но между ними есть различия. Основные различия заключаются в том, что по мере продвижения алгоритм обновляет стоимость достижения каждой исследованной вершины от начальной вершины. Кроме того, при выборе маршрута A^* рассматривает кратчайший путь от начала до цели через текущую вершину, поскольку суммирует оценку эвристики со стоимостью пути, вычисленной до текущей вершины. Этот процесс позволяет алгоритму выполнять больше вычислений, чем поиск в ширину, когда эвристика является правильной оценкой, и находить наилучший возможный путь.



СОВЕТ

Поиск кратчайшего возможного пути с точки зрения стоимости — это основная функция алгоритма Дейкстры. A^* — это просто алгоритм Дейкстры, в котором стоимость достижения вершины дополняется эвристической оценкой ожидаемого расстояния до цели. Алгоритм Дейкстры подробно описан в главе 9. Повторное обращение к материалу главы 9 поможет вам лучше понять, как A^* использует эвристики.

```

# A*
open_list = set(graph.nodes())
closed_list = {start: manhattan_dist(
    start, goal, coordinates)}
visited = {start: 0}
path = {}

while open_list:

    candidates = open_list&closed_list.keys()
    if len(candidates)==0:
        print ("Cannot find a way to the goal %s" % goal)
        break
    frontier = [(closed_list[node],
        node) for node in candidates]
    score, min_node =sorted(frontier)[0]

    if min_node==goal:
        print ("Arrived at final vertex %s" % goal)
        print ('Unvisited vertices: %i' % (len(
            open_list)-1))
        break
    else:
        print("Processing vertex %s, " % min_node, end="")

    open_list = open_list.difference(min_node)
    current_weight = visited[min_node]
    neighbors = node_neighbors(graph, min_node)
    to_be_visited = list(neighbors-visited.keys())
    for node in neighbors:
        new_weight = current_weight + graph_weight(
            graph, min_node, node)
        if node not in visited or \
        new_weight < visited[node]:
            visited[node] = new_weight
            closed_list[node] = manhattan_dist(node, goal,
                coordinates) + new_weight
            path[node] = min_node

    if to_be_visited:
        print ("discovered %s" % to_be_visited)
    else:
        print ("getting back to open list")

```

```
print ('\nBest path is:', reconstruct_path(
    path, start, goal))
print ('Length of path: %i' % compute_path_dist(
    reconstruct_path(path, start, goal), graph))
```

Когда A^* завершает анализ лабиринта, он выдает оптимальный путь, который короче решения, найденного поиском в ширину:

```
Processing vertex A, discovered ['F', 'B', 'G']
Processing vertex B, discovered ['C']
Processing vertex F, discovered ['L', 'K']
Processing vertex G, discovered ['H']
Processing vertex C, getting back to open list
Processing vertex K, discovered ['Q', 'P']
Processing vertex H, discovered ['M']
Processing vertex L, discovered ['R']
Processing vertex P, discovered ['U', 'V']
Processing vertex M, discovered ['N']
Processing vertex Q, discovered ['W']
Processing vertex R, discovered ['S']
Processing vertex U, getting back to open list
Processing vertex N, discovered ['J']
Processing vertex V, getting back to open list
Processing vertex S, discovered ['Y', 'X', 'O']
Processing vertex W, getting back to open list
Processing vertex X, discovered ['T']
Processing vertex J, discovered ['E']
Arrived at final vertex Y
Unvisited vertices: 5

Best path is: ['A', 'F', 'L', 'R', 'S', 'Y']
Length of path: 13
```

У этого решения есть своя цена: A^* исследует почти все имеющиеся вершины, оставляя неисследованными всего пять вершин. Как и у алгоритма Дейкстры, его наихудшее время выполнения составляет $O(v^2)$, где v — количество вершин в графе; или $O(e + v * \log(v))$, где e — количество ребер, при использовании очередей с минимальным приоритетом — эффективной структуры данных, когда нужно получить минимальное значение для длинного списка. Алгоритм A^* не отличается по наихудшему времени выполнения от алгоритма Дейкстры, хотя в среднем работает лучше на больших графах, поскольку быстрее находит целевую вершину при правильном направлении эвристической метрикой (в случае маршрутизирующего робота — манхэттенским расстоянием).



В ЭТОМ РАЗДЕЛЕ

Разработка продвинутых методов обработки данных

Более широкое использование автоматизации и автоматических откликов

Более быстрое выполнение задач по обработке данных

Обеспечение того, чтобы алгоритмы вели себя должным образом

В ЭТОЙ ГЛАВЕ

- » Рассмотрение процедур сортировки и поиска
- » Использование случайных чисел
- » Уменьшение объема данных
- » Обеспечение сохранности данных и многое другое...

ГЛАВА 21

Десять алгоритмов, которые меняют мир

Сложно представить, что алгоритм может делать что-то физическое, не говоря уже о том, чтобы менять мир. Однако сегодня алгоритмы встречаются повсюду, и вы можете даже не осознавать, как сильно они влияют на вашу жизнь. Большинство людей понимает, что интернет-магазины и другие торговые площадки используют алгоритмы для подбора сопутствующих товаров на основе предыдущих покупок. Однако мало кто знает о применении алгоритмов в медицине — многие из них помогают врачам ставить диагнозы.

Алгоритмы встречаются в самых неожиданных местах. Время переключения светофоров часто зависит от алгоритмических расчетов. Алгоритмы помогают вашему смартфону общаться с вами, а телевизору — делать то, чего не могли телевизоры прошлого. Поэтому неудивительно, что алгоритмы готовы изменить мир. В этой главе рассматриваются десять таких алгоритмов.



ЗАПОМНИТЕ

Для пуристов можно сказать, что алгоритмы меняли мир на протяжении столетий, так что по сути ничего не изменилось за тысячи лет. Вавилоняне использовали алгоритмы для факторизации и нахождения квадратных корней еще в 1800–1600 годах до нашей эры (<https://www.historyofinformation.com/detail.php?entryid=4379>). Аль-Хорезми описал алгоритмы решения линейных и квадратных уравнений около 820 года нашей эры (<https://www.newscientist.com/people/muhammad-ibn-musa-al-khwarizmi>). Эта глава посвящена компьютерным алгоритмам, но сами алгоритмы существуют уже очень давно.

§ 1. Использование процедур сортировки

Без упорядоченных данных бóльшая часть мира остановилась бы. Чтобы использовать данные, нужно уметь их находить. Сотни алгоритмов сортировки описаны на таких сайтах, как <https://betterexplained.com/articles/sorting-algorithms>, а также в этой книге (см. главу 7).

Однако три наиболее распространенных алгоритма сортировки — это сортировка слиянием, быстрая сортировка и пирамидальная сортировка, благодаря их превосходной скорости (сравнение времени выполнения можно найти на <https://www.cprogramming.com/tutorial/computersciencetheory/sortcomp.html>). Выбор наиболее подходящего алгоритма сортировки для вашего приложения зависит от следующих факторов:

- » ожидаемых задач приложения;
- » типа обрабатываемых данных;
- » доступных вычислительных ресурсов.

Суть в том, что возможность сортировать данные в любую форму, необходимую приложению для выполнения задачи, заставляет мир работать, и эта возможность меняет то, как устроен мир.

Некоторые современные компании процветают благодаря использованию алгоритмов сортировки. Например, Google существует потому, что помогает людям находить информацию, и эта способность во многом основана на возможности сортировать данные, делая их легкодоступными. Представьте, насколько сложно было бы найти товар на Amazon без алгоритма сортировки. Даже приложение с рецептами на вашем домашнем компьютере активно использует процедуры сортировки для поддержания порядка в данных. По сути, не будет преувеличением сказать, что любое серьезное приложение в значительной степени опирается на алгоритмы сортировки.

§ 2. Поиск вещей с помощью процедур поиска

Как и алгоритмы сортировки, поисковые алгоритмы сегодня присутствуют практически в каждом крупном приложении. Они встречаются повсюду, даже там, где вы могли бы об этом не задумываться, например в вашем автомобиле. Быстрый поиск информации стал неотъемлемой частью повседневной жизни. Представьте, например, что вы опаздываете на встречу и вдруг обнаруживаете, что ваш GPS не может найти нужный адрес. Как и алгоритмы сортировки, поисковые алгоритмы бывают самых разных видов и типов, и их описание можно найти на таких сайтах, как <https://tekmarathon.com/2012/10/05/best-searching-algorithm-2> и <https://www.geeksforgeeks.org/searching-algorithms>. На самом деле, поисковых алгоритмов даже больше, чем алгоритмов сортировки, поскольку требования к поиску часто бывают более строгими и сложными. Многие поисковые алгоритмы тоже рассматриваются в этой книге (см. главу 7).

§ 3. Встряхиваем вещи с помощью случайных чисел

Многие вещи были бы гораздо менее увлекательными без элемента случайности. Например, представьте, что при каждом запуске пасьянса «Солитер» вы видите в точности одну и ту же раскладку карт. Никто не стал бы играть в такую игру. Поэтому генерация случайных чисел — важнейшая часть игрового процесса. Более того, как описано в нескольких главах этой книги, некоторым алгоритмам для правильной работы необходим определенный уровень случайности (см. блок «Рассмотрим, почему нужна рандомизация» в главе 17). Вы также обнаружите, что в некоторых случаях тестирование работает лучше при использовании случайных значений (см. блок «Выбор конкретного типа сжатия» в главе 14).



ЗАПОМНИТЕ

Числа, которые вы получаете от алгоритма, обычно генерируются с помощью генераторов псевдослучайных чисел (PRNG), что означает потенциальную возможность предсказать следующее число в последовательности, зная алгоритм и начальное значение, использованное для генерации числа. Именно поэтому эта информация так тщательно охраняется. Существует возможность установить на компьютер аппаратное обеспечение для создания генератора истинно случайных чисел (TRNG), как описано

в статье «Генерация случайных чисел намного сложнее, чем вы думаете» по адресу <https://betterprogramming.pub/generating-random-numbers-is-a-lot-harder-than-you-think-b121c3e75d08>. К сожалению, TRNG обладают некоторыми недостатками (например, в некоторых случаях они могут создавать небезопасную криптографию). Поэтому в статье также описывается криптографически стойкий генератор псевдослучайных чисел (CSPRNG). Иными словами, вы можете столкнуться с различными методами генерации случайных чисел, у каждого из которых своя целевая область применения.



ПРИМЕЧАНИЕ

Аппаратные методы генерации случайных чисел могут основываться на атмосферных шумах или изменениях температуры (подробнее см. на <https://engineering.mit.edu/engage/ask-an-engineer/can-a-computer-generate-a-truly-random-number>). Более того, вы можете приобрести аппаратное решение для генерации случайных чисел, например TrueRNG (<https://www.amazon.com/exec/obidos/ASIN/B01KR2JHTA/datacservip0f-20>) или ChaosKey (<https://altusmetrum.org/ChaosKey>) (только схема), и подключить его к USB-порту для генерации того, что, вероятно, является действительно случайными числами. Интересная особенность сайта ChaosKey: там представлена схема, показывающая, как устройство собирает случайный шум и преобразует его в случайное число.

§ 4. Выполнение сжатия данных

В главе 14 рассматриваются методы сжатия данных и используются те виды сжатия, которые обычно применяются для файлов. Однако сжатие данных сегодня влияет на все аспекты компьютерных технологий. Например, большинство графических, видео- и аудиофайлов полагается на сжатие данных. Без сжатия данных было бы невозможно достичь необходимого уровня пропускной способности, чтобы обеспечить работу таких задач, как потоковое видео.

Тем не менее сжатие данных находит даже больше применений, чем можно было бы ожидать. Практически каждая система управления базами данных (СУБД) использует сжатие данных, чтобы информация занимала разумный объем на диске. Облачные вычисления были бы невозможны без сжатия данных, поскольку загрузка файлов из облака на локальные машины занимала бы слишком много времени. Даже веб-страницы часто полагаются на сжатие данных для передачи информации из одного места в другое.

§ 5. Сохранение данных в секрете

Идея сохранения данных в тайне не нова. Фактически это одна из старейших причин использования алгоритмов. Слово *криптография* происходит от двух греческих слов: *kryptós* («скрытый» или «тайный») и *graphein* («письмо»). Считается, что греки были первыми, кто начал использовать криптографию (см. на <https://www.britannica.com/topic/cipher>), а древние тексты свидетельствуют о том, что Юлий Цезарь использовал зашифрованные послания для общения со своими генералами. Суть в том, что сохранение данных в тайне — одна из самых длительных битв в истории. Как только одна сторона находит способ сохранить секрет, кто-то другой находит способ сделать этот секрет достоянием общественности, взломав криптографическую защиту.

Основные направления использования компьютерной криптографии сегодня включают:

- » **конфиденциальность** — обеспечение того, чтобы никто не мог видеть информацию, которой обмениваются две стороны;
- » **целостность данных** — снижение вероятности того, что кто-то или что-то может изменить содержание данных, передаваемых между двумя сторонами;
- » **аутентификацию** — определение личности одной или нескольких сторон;
- » **неотказуемость** — снижение возможности для стороны заявить, что она не совершала определенного действия.



СОВЕТ

Помимо сохранения секретности при использовании компьютеров, история компьютерных криптографических алгоритмов длинна и увлекательна. Параграф «Соккрытие ваших секретов с помощью криптографии» в главе 14 дает хороший обзор криптографических методов. Список часто используемых алгоритмов (как современных, так и исторических) можно найти на <https://www.axel.org/2021/05/28/history-of-encryption> и <https://komodoplatform.com/en/academy/history-of-cryptology>. Руководство на https://www.owasp.org/index.php/Guide_to_Cryptography представляет дополнительные подробности о принципах работы криптографии.

§ 6. Изменение домена данных

Преобразование Фурье и быстрое преобразование Фурье (БПФ) играют огромную роль в том, как приложения воспринимают данные. Эти два алгоритма преобразуют данные из частотной области (как быстро осциллирует сигнал) во временную область (временная разница между изменениями сигнала). Фактически невозможно получить какую-либо степень в области компьютерного оборудования, не потратив значительного времени на работу с этими двумя алгоритмами. Время — это всё.



ЗАПОМНИТЕ

Зная частоту изменений, можно определить временной интервал между ними и, следовательно, понять, сколько времени есть на выполнение задачи до того, как изменение состояния потребует других действий. Такие алгоритмы широко применяются в различных фильтрах. Без их фильтрующего эффекта было бы невозможно достоверно воспроизводить видео и аудио через потоковое соединение. Все эти приложения звучат довольно сложно, и так оно и есть, но существуют замечательные руководства, которые помогают лучше понять принцип работы этих алгоритмов (например, см. руководство по адресу <https://w.astro.berkeley.edu/~jrg/ngst/fft/fft.html>). Руководство на <https://betterexplained.com/articles/an-interactive-guide-to-the-fourier-transform> тоже очень интересно и увлекательно, особенно если вам нравятся смузи.

§ 7. Анализ ссылок

Способность анализировать взаимосвязи — это то, что сделало современные вычисления уникальными. Возможность сначала создавать представление этих взаимосвязей, а затем анализировать их — тема третьего раздела этой книги. По сути, вся идея веба заключается в создании связей, и связность учитывалась с самого начала того, что стало всемирным феноменом. Без возможности анализировать и использовать связи не работали бы такие приложения, как базы данных и электронная почта. Вы не смогли бы эффективно общаться с друзьями в социальных сетях.

По мере развития веба и того, как люди все больше осваивались с устройствами, делающими связность более простой и повсеместной, социальные сети и торговые площадки вроде Amazon стали активнее использовать анализ связей, чтобы, например, продавать вам больше товаров.

Разумеется, анализ связей делает больше, чем просто информирует в контексте взаимосвязей. Подумайте об использовании анализа связей для предоставления маршрутов движения или поиска причинно-

следственных связей между деятельностью человека и заболеваниями. Анализ связей позволяет увидеть взаимосвязь вещей, о которых вы обычно не задумываетесь, но которые влияют на вашу повседневную жизнь. Благодаря анализу связей вы можете прожить дольше, поскольку врач может посоветовать вам, какие привычки следует изменить, чтобы исправить проблемы, которые могут возникнуть позже. Суть в том, что связи повсюду, а анализ связей предлагает метод определения того, где эти связи существуют и действительно ли они важны.

§ 8. Выявление закономерностей в данных

Данные не существуют в вакууме. На них влияют самые разные факторы, включая предвзятость, которая окрашивает то, как люди воспринимают информацию. В главе 10 рассматривается тенденция данных группироваться в определенных средах и анализ этих кластеров, который может рассказать вам много интересного о самих данных.



СОВЕТ

Анализ шаблонов находится на переднем крае некоторых самых удивительных применений компьютеров сегодня. Например, фреймворк обнаружения объектов Виолы — Джонса (подробнее см. на <https://towardsdatascience.com/understanding-face-detection-with-the-viola-jones-object-detection-framework-c55cc2a9da14>) делает возможным распознавание лиц в реальном времени. Этот алгоритм может помочь повысить безопасность в таких местах, как аэропорты, где в настоящее время действуют злоумышленники. Подобные алгоритмы могут помочь вашему врачу обнаружить различные виды рака задолго до того, как опухоль станет видимой человеческому глазу. Более раннее обнаружение повышает вероятность полного выздоровления. То же самое относится и ко многим другим медицинским проблемам (например, к обнаружению переломов костей, которые слишком малы, чтобы их увидеть, но все же вызывают боль).

Распознавание шаблонов также используется для более приземленных целей. Например, анализ шаблонов позволяет людям обнаруживать потенциальные проблемы с дорожным движением до их возникновения. Возможно также использовать анализ шаблонов, чтобы помочь фермерам выращивать больше продовольствия при меньших затратах, применяя воду и удобрения, только когда это необходимо. Использование распознавания шаблонов также может помочь управлять дронами на полях, что делает работу фермера эффективнее по времени и позволяет обрабатывать больше земли при меньших затратах. Без алгоритмов такие шаблоны, которые столь значительно влияют на повседневную жизнь, не могут быть распознаны.

§ 9. Работа с автоматизацией и автоматическими ответами

Пропорционально-интегрально-дифференциальный алгоритм — довольно сложное название. Попробуйте просто произнести это три раза быстро! Тем не менее это один из важнейших секретных алгоритмов, о котором вы никогда не слышали, но на который полагаетесь каждый день. Подробности об этом алгоритме можно найти по адресу <https://www.ni.com/en-us/innovations/white-papers/06/pid-theory-explained.html>, включая блок-схему, показывающую один из возможных вариантов реализации.



ЗАПОМНИТЕ

Пропорционально-интегрально-дифференциальный алгоритм использует механизм обратной связи в контуре управления для минимизации ошибки между желаемым выходным сигналом и реальным выходным сигналом. Алгоритм можно встретить повсюду в управлении автоматизацией и автоматическими реакциями. Например, когда ваш автомобиль входит в занос из-за слишком резкого торможения, этот алгоритм помогает обеспечить правильную работу антиблокировочной системы (АБС). В противном случае АБС могла бы излишне среагировать и только ухудшить ситуацию.

Практически все современные механизмы используют пропорционально-интегрально-дифференциальный алгоритм. По сути, без него робототехника была бы невозможна. Представьте, что случилось бы на заводе, если бы все роботы постоянно избыточно реагировали на каждое выполняемое действие. Возникший хаос быстро убедил бы владельцев вообще отказаться от использования машин для любых целей.

§ 10. Создание уникальных идентификаторов

Кажется, будто каждый человек — это просто номер. Вернее, не один номер, а множество номеров. У людей есть номера кредитных карт, номера водительских удостоверений и государственных идентификаторов. То же самое касается всевозможных компаний и организаций. Людям приходится вести списки всех этих номеров, потому что их просто слишком много, чтобы удержать в памяти. При этом каждый из этих номеров должен однозначно идентифицировать человека для определенной стороны. За всей этой уникальностью стоят различные алгоритмы.

В главе 7 обсуждаются хеши — один из способов обеспечения уникальности. В основе как хеширования, так и криптографии лежит факторизация целых чисел — тип алгоритма, который раскладывает очень большие числа на простые множители. Фактически факторизация целых чисел — одна из самых сложных алгоритмических задач, но люди постоянно работают над этой проблемой. Современное общество настолько зависит от возможности однозначной идентификации личности, что скрытые секреты создания таких идентификаторов стали неотъемлемой частью современного мира.

В ЭТОЙ ГЛАВЕ

- » Выполнение задач быстрее и лучше
- » Решение новых типов задач
- » Рассмотрение осуществимости гиперкомпьютеров
- » Использование односторонних функций и многое другое...

ГЛАВА 22

Десять нерешенных алгоритмических проблем

Алгоритмы действительно существуют уже столетия, поэтому можно было бы подумать, что ученые уже открыли и усовершенствовали все возможные алгоритмы. К сожалению, всё наоборот. Решение задачи с помощью конкретного алгоритма часто поднимает новые вопросы, которые алгоритм не решает и которые не были очевидны, пока кто-то не нашел решение. Кроме того, развитие технологий и изменение образа жизни постоянно рожают новые вызовы, требующие создания новых алгоритмов. Например, растущая взаимосвязанность общества и использование роботов увеличили потребность в новых алгоритмах.

Как было показано в главе 1, алгоритмы представляют собой последовательность шагов для решения задачи, и их не следует путать с другими сущностями, например уравнениями. Алгоритм никогда не бывает решением в поисках проблемы. Никто не станет создавать последовательность шагов для решения задачи, которая еще не существует (или может никогда не возникнуть). Более того, многие проблемы интересны, но не требуют срочного решения. Поэтому, даже если все знают о проблеме и понимают, что кому-то может понадобиться ее решение, никто не торопится его создавать.

Эта глава посвящена алгоритмическим задачам, решение которых было бы полезным, если бы кто-то его нашел. Короче говоря, эта глава важна потому, что вы можете найти проблему, которую действительно хотели бы решить, и даже захотеть стать частью команды, которая найдет решение.

§ 1. Быстрое решение проблем

По мере развития машинного обучения и растущей зависимости людей от компьютеров в решении задач вопрос о том, как быстро компьютер может решить проблему, становится критически важным. Проблема «Р против NP» просто спрашивает, может ли компьютер быстро решить задачу, если он может быстро проверить ее решение (подробнее см. в блоке «Рассмотрение NP-полных задач» главы 15). Другими словами, если компьютер может за полиномиальное время или быстрее удостовериться в правильности человеческого решения задачи, может ли он сам решить эту задачу за полиномиальное время или быстрее?

Джон Нэш впервые обсудил этот вопрос в 1950-х годах в письмах в Агентство национальной безопасности (АНБ), и эта дискуссия возникла снова в переписке между Куртом Гёделем и Джоном фон Нейманом. Помимо машинного обучения (и ИИ в целом), эта проблема важна для многих других областей, включая математику, криптографию, исследование алгоритмов, теорию игр, обработку мультимедиа, философию и экономику.

§ 2. Более эффективное решение задач 3SUM

Задача 3SUM ставит вопрос: для заданного набора действительных чисел существуют ли три числа, сумма которых равна нулю? Задачу можно обобщить до k чисел для особых случаев, превратив ее в задачу k -SUM. На первый взгляд, 3SUM не кажется проблемой, достойной решения. Она обычно используется в вычислительной геометрии для решения таких задач, как:

- » определение наличия трех *коллинеарных точек* (точек, лежащих на одной прямой) в наборе точек;
- » вычисление площади объединения набора треугольников;
- » определение возможности размещения одного выпуклого многоугольника внутри другого.

Однако в реальной жизни часто возникают ситуации с тремя участниками, где сумма их взаимодействий должна быть равна нулю (как в игре с нулевой суммой). Простой пример — посредническая финансовая операция. Еще более актуальный пример — обмен углеродными квотами

для достижения углеродной нейтральности, который включает обмен выбросами и квотами между компаниями или странами для поддержания стабильного уровня выбросов углекислого газа.

Хотя задачу 3SUM можно легко решить за время $O(n^2)$, различные специалисты в области информатики работали над снижением временной сложности решения этого алгоритма. После различных улучшений текущий лучший алгоритм может решить задачу 3SUM за время $O(n^2(\log \log n)^{O(1)/\log 2})$, однако ученые полагают, что теоретически эту границу можно снизить еще больше.

§ 3. Ускорение умножения матриц

Умножение матриц сейчас используется повсеместно для решения широкого спектра задач. Проблема в том, что умножение матриц происходит медленно — действительно медленно, — поскольку требует множества шагов. Согласно статье на сайте Quanta Magazine (<https://www.quantamagazine.org/mathematicians-inch-closer-to-matrix-multiplication-goal-20210323/>), текущая цель — достичь количества шагов умножения порядка n^2 . (Стандартный подход из учебников математики требует n^3 шагов.) Достижение этой цели означало бы, что для умножения матрицы 2×2 потребуется четыре шага вместо восьми шагов умножения (плюс несколько сложений), которые требуются сейчас. Согласно OpenGenus (<https://iq.opengenus.org/unsolved-problems-in-algorithms/>), умножение матриц — одна из главных нерешенных проблем в области алгоритмов.

Так где же используется умножение матриц? Многие применения — это то, что любят математики и графические дизайнеры, например искажение Моны Лизы, чтобы она выглядела странно (см. статью на betterexplained.com). Обсуждение на Quora (<https://www.quora.com/What-are-some-of-the-real-time-applications-of-Matrix-multiplication>) дает представление о некоторых полезных практических применениях, например о том, где разместить новую пожарную часть.

§ 4. Определение остановки программы

Одна из проблем, предложенных Аланом Тьюрингом в 1936 году, заключается в вопросе о том, может ли алгоритм, получив описание программы и входные данные, определить, остановится ли эта программа (*проблема остановки*). При работе с простым приложением во многих случаях легко определить, остановится программа или продолжит работать в бесконечном цикле. Однако по мере увеличения сложности программы определить результат ее выполнения с любыми входными данными становится все труднее. Машина Тьюринга не может сделать такое определение; результатом становится некорректный код с бесконечными циклами. Никакое тестирование с использованием современных технологий не может решить эту проблему.



ЗАПОМНИТЕ

Гиперкомпьютер — это вычислительная модель, которая выходит за рамки машины Тьюринга и способна решать такие задачи, как проблема остановки. Однако создание таких машин невозможно при текущем уровне технологий. Если бы они существовали, с их помощью можно было бы получить ответы на всевозможные неразрешимые вопросы, с которыми не справляются современные компьютеры. Статья по адресу <https://www.newscientist.com/article/mg22329781-500-what-will-hypercomputers-let-us-do-good-question> дает хорошее представление о том, что произойдет, если кто-то сможет решить эту проблему. О возможной будущей реализации можно прочитать здесь: <https://www.foprc.org/quantum-hypercomputing-by-means-of-evanescent-photons.php>.

§ 5. Создание и использование односторонних функций

Односторонняя функция — это функция, которую легко использовать для получения ответа в одном направлении, но практически невозможно применить к обратному значению этого ответа. Другими словами, односторонняя функция используется для создания чего-то вроде хеша, который может быть частью решения задач криптографии, идентификации личности, аутентификации или других потребностей в области защиты данных.



Существование односторонней функции — это не столько загадка, сколько вопрос доказательства. Многие системы телекоммуникации, электронной коммерции и электронного банкинга в настоящее время полагаются на функции, которые предположительно односторонние, но никто на самом деле не знает, действительно ли они таковыми являются. Существование односторонней функции остается гипотезой, а не теорией (объяснение разницы между ними можно найти по адресу https://www.diffen.com/difference/Hypothesis_vs_Theory). Если бы кто-то смог доказать существование односторонней функции, это значительно упростило бы решение проблемы безопасности данных с точки зрения программирования.

§ 6. Умножение действительно больших чисел

Очень большие числа встречаются во многих областях. Например, рассмотрим вычисления, связанные с расстояниями до Марса или, скажем, Плутона. Существуют методы умножения очень больших чисел, но они, как правило, работают медленно, поскольку требуют выполнения множества операций. Проблема возникает, когда числа слишком велики, чтобы поместиться в регистрах процессора. В этом случае умножение должно выполняться в несколько этапов, что значительно замедляет процесс. Существующие решения включают:

- » алгоритм комплексного умножения Гаусса;
- » умножение Карацубы;
- » метод Тоома – Кука;
- » методы преобразования Фурье.

Хотя многие из доступных сегодня методов дают приемлемые результаты, все они требуют времени, а когда нужно выполнить большое количество вычислений, проблема времени может стать критической. Следовательно, умножение больших чисел — одна из тех задач, которая требует лучшего решения, чем те, что доступны сегодня.

§ 7. Разделение ресурсов поровну

Разделить ресурсы поровну может показаться несложным, но люди, будучи завистливыми созданиями, могут посчитать распределение несправедливым, если не найти способ убедить каждого в честности разделения. Это и есть задача о справедливом разрезании торта без зависти. Когда вы разрезаете торт, как бы справедливо вы ни старались это сделать, всегда возникает ощущение неравномерности деления. Создание справедливого распределения ресурсов важно в повседневной жизни для минимизации конфликтов между заинтересованными сторонами в любой организации, что делает работу всех более эффективной.



СОВЕТ

Для задачи о разрезании торта без зависти уже существует два решения с определенным числом участников, но общего решения пока нет. Когда участвуют два человека, первый разрезает торт, а второй выбирает первый кусок. При таком способе оба участника уверены в равном делении. С тремя людьми задача усложняется, но существует решение Селфриджа — Конвея, которое можно найти по адресу <https://archive.ochronus.com/cutting-the-pie> (на сайте обсуждается пирог, но принцип тот же). Однако когда число участников достигает четырех, решения не существует.

§ 8. Сокращение времени расчета расстояния редактирования

Редакционное расстояние между двумя строками — это количество операций, необходимых для преобразования одной строки в другую. Расчет расстояния основан на операциях расстояния Левенштейна, которые включают удаление, вставку или замену символа в строке. Эта техника применяется в естественно-языковых интерфейсах, количественном анализе последовательностей ДНК и во многих других областях, где требуется сравнение или модификация похожих строк.

Существует несколько решений этой задачи, и все они довольно медленные. Фактически у большинства из них есть временная сложность $O(n^2)$, поэтому время, необходимое для выполнения преобразования, быстро возрастает до такой степени, что люди замечают паузы при обработке ввода. Пауза не так заметна при использовании текстового редактора,

который выполняет автоматическую проверку слов и исправляет неправильно написанные слова. Однако при использовании голосовых интерфейсов пауза может стать весьма ощутимой и привести к ошибкам со стороны пользователя. Текущая цель — добиться вычисления редакционного расстояния за субквадратичное время: $O(n^2)$.

§ 9. Игра в игру «Паритет»

На первый взгляд может показаться, что решение игры не так уж полезно в реальной жизни. Да, игры забавны и интересны, но, согласно общепринятому мнению, они не дают основы для чего-то действительно полезного. Однако теория игр находит применение в многих реальных сценариях, включающих сложные процессы, которые людям проще понять именно как игры, а не как собственно процессы. В данном случае игра помогает людям понять автоматическую верификацию и синтез контроллеров, среди прочего. Подробнее об игре на четность можно прочитать по адресу <http://www.sciencedirect.com/science/article/pii/S0890540115000723>. Вы даже можете поиграть в нее на <https://www.abefehr.com/parity>.

§ 10. Понимание пространственных проблем

Чтобы понять контекст этой конкретной проблемы, представьте перемещение коробок на складе или другие ситуации, где нужно учитывать пространство, в котором происходит движение. Очевидно, если у вас много коробок на большом складе и для их перемещения нужен погрузчик, вы не захотите пытаться определить оптимальный способ их хранения путем физической перестановки. Здесь вам нужно решить проблему, визуализируя решение.

Однако вопрос в том, есть ли у всех пространственных задач решение. В этом случае подумайте об одной из детских головоломок, где нужно собрать картинку, передвигая маленькие плитки. Кажется, что решение должно существовать во всех случаях, но иногда неудачная начальная позиция может привести к ситуации, не имеющей решения. Обсуждение такой проблемы можно найти по адресу <https://math.stackexchange.com/questions/754827/does-a-15-puzzle-always-have-a-solution>.



СОВЕТ

Математики, такие как Сэм Лойд (см. на <https://www.mathsisfun.com/puzzles/sam-loyd-puzzles-index.html>), часто используют головоломки для демонстрации сложных математических задач; у некоторых из них нет решения и сегодня. Посещение этих сайтов увлекательно, поскольку вы не только получаете бесплатное развлечение, но и пищу для размышлений. Вопросы, которые поднимают эти головоломки, имеют практическое применение, но представлены в увлекательной форме.

Об авторах

Джон Пол Мюллер — внештатный автор и технический редактор. Писательство у него в крови: на сегодняшний день он написал 121 книгу и более 600 статей. Темы его работ охватывают самые разные области — от сетевых технологий до искусственного интеллекта, от управления базами данных до программирования с полной отдачей. Среди его текущих книг есть работы, посвященные науке о данных, машинному обучению и алгоритмам. Он также пишет о языках программирования, таких как C++, C и Python. Его навыки технического редактирования помогли более чем 70 авторам улучшить содержание их рукописей. Джон оказывал услуги технического редактирования различным журналам, занимался разнообразной консультационной деятельностью и составляет сертификационные экзамены. Обязательно читайте блог Джона по адресу <http://blog.johnmuellerbooks.com>. Связаться с Джоном можно по электронной почте John@JohnMuellerBooks.com. У Джона также есть веб-сайт <http://www.johnmuellerbooks.com>.

Лука Массарон — специалист в обработке данных и директор по маркетинговым исследованиям, специализирующийся на многомерном статистическом анализе, машинном обучении и анализе потребительского поведения. У него более десяти лет опыта решения реальных задач и создания ценности для заинтересованных сторон путем применения логического мышления, статистики, интеллектуального анализа данных и алгоритмов. От роли пионера в области анализа веб-аудитории в Италии до достижения позиции в первой десятке участников на платформе kaggle.com — он всегда был увлечен всем, что связано с данными и аналитикой, и стремился продемонстрировать потенциал основанного на данных получения знаний как экспертам, так и новичкам. Отдавая предпочтение простоте перед ненужным усложнением, он считает, что в науке о данных многого можно достичь, понимая и применяя ее основы. Лука также является экспертом по машинному обучению программы Google Developer Expert (GDE).

Посвящение Джона

Эта книга посвящается всем людям, которые поддерживали меня во время различных перемен в моей жизни.

Посвящение Луки

Я посвящаю эту книгу моей жене Юкико, которую я всегда нахожу любознательной и готовой удивляться чудесам этого удивительного и озадачивающего технологического мира. С любовью.

Благодарности Джона

Спасибо моей жене Ребекке. Хотя ее уже нет с нами, ее дух живет в каждой книге, которую я пишу, и в каждом слове, появляющемся на странице. Она верила в меня, когда никто другой не верил.

Род Стивенс заслуживает благодарности за техническое редактирование этой книги. Он значительно повысил точность и глубину материала, который вы здесь видите. Критическое мышление Рода заставляет меня действительно тщательно продумывать все элементы книги. Он также выступает в роли проверяющего здравого смысла для моей работы.

Мэтт Вагнер, мой агент, заслуживает признания за помощь в получении контракта и заботу обо всех деталях, о которых большинство авторов даже не задумываются. Я всегда ценю его поддержку. Приятно знать, что кто-то хочет помочь.

Некоторые люди читали эту книгу полностью или частично, помогая мне усовершенствовать подход, протестировать примеры кода и в целом предоставить тот вклад, который хотели бы внести все читатели. Эти добровольцы помогали самыми разными способами, которые невозможно здесь перечислить. Я особенно признателен Еве Битти, которая давала общие рекомендации, прочитала всю книгу и самоотверженно посвятила себя этому проекту.

Наконец, я хотел бы поблагодарить Келси Бэрд, Сюзан Кристоферсен, Мишель Хакер и остальных сотрудников редакции и производственного отдела.

Благодарности Луки

Моя самая большая благодарность — моей семье, Юкико и Амелии, за их поддержку, жертвы и любящее терпение во время долгих дней и ночей, недель и месяцев работы над этой книгой.

Я благодарю всех сотрудников редакции и производственного отдела Wiley, особенно Келси Бэрд и Сьюзан Кристоферсен, за их высокий профессионализм и поддержку на всех этапах написания этой книги серии *For Dummies*.

Изображение на обложке: © Kobol75/Shutterstock

Предметный указатель

- 7-Zip, 309
- AGraph, 178, 179
- Amazon Web Services с Elastic MapReduce (EMR), 293
- American Standard Code for Information Interchange (ASCII), 127
- Apache Lucene, 290
- Apache Nutch, 290
- Apache Software Foundation, 290
- BFS (breadth-first search), 193
- Colab (Google), 15
 - аппаратное ускорение, 79
 - блокнот, 69, 71
 - выполнение общих задач, 74
 - знакомство с возможностями, 65
 - определение, 63
 - перемещение ячеек, 78
 - получение помощи, 80
 - работа с Colab, 62, 64
 - редактирование ячеек, 78
 - создание ячеек, 75, 77
 - сохранение, 72
 - сравнение файлов, 68
- CPLEX, 421
- DFS (depth-first search), 195
- DjVu, 308
- Fairchild Semiconductor, 258
- Firefox, 64
- $G = (V, E)$, 171
- Gartner, 264
- GitHub, 64, 70
- Google Colaboratory, 62
- Google DeepMind, 336
- Google Drive, 70
- Google PageRank, 237, 240, 243, 244, 249, 253
- GPS, 28, 190
- Hadoop, 290
- HITS (Hypertext-Induced Topic Search), 242
- HyperLogLog, 281
- IBM Almaden, 242
- IBM Research, 339
- Intel, 258
- JPEG, 308
- Jupyter Notebook, 15
- LHA, 309
- LogLog, 281
- MapReduce, 290, 293
- matplotlib, 178
- Matrix Reshish, 119
- MAX-3SAT, 429
- MP3, 308
- MPEG, 308
- MrJob, 293
- NetworkX, 178
- NP-полная задача, 388
- NP-трудная задача, 363
- NumPy, 84, 134, 422
- PageRank, 240
- Pandas, 129, 132, 134, 370
- Project Gutenberg, 298
- PuLP, 412, 422
- PyCrypto, 324
- Python, 37, 62, 69
 - алгоритм Дейкстры, 216
 - борьба с повторениями, 94
 - вычисления, 83
 - жадный подход, 331
 - лямбда-функция, 294
 - методы шифрования, 324
 - очереди, 134
 - почему стоит использовать классы, 107
 - рекурсия, 95
 - скалярные и векторные операции, 84
 - словари, 135
 - фильтр Блума, 278
 - хранение данных, 132
- RAM-компьютер, 56
- RandomWalkSAT, 405
- RankBrain, 254
- RAR, 309

Raspberry Pi, 34
 RLE, 314
 Roomba, 373
 SciPy, 186
 Sentinel-1A, 263
 SEO-оптимизация, 240
 SEO-специалисты, 240
 TrueRNG, 454
 UTF-8 (Unicode Transformation Format 8), 307
 WMA, 308
 XPRESS, 421
 ZIP, 309
 абстрактные машины, 56
 абсцисса (координата x), 593
 авторитетные страницы, 242
 агент, 33
 Адамашек, Анна, 51
 Адамашек, Михал, 51
 алгоритмы
 A*, 427, 428, 436
 MD5 от RSA, 164
 Pledge, 437
 анализ, 40
 Атлантик-Сити, 375
 безопасного хеширования (SHA), 164
 Беллмана – Форда, 211
 Борувки, 203
 быстрого выбора, 372
 важность точности, 186
 внедрение в бизнес, 264
 выбор правильных, 203
 генетические, 414, 430
 данные для получения выводов, 34
 Дейкстры, 211
 евклидово расстояние, 435
 жадный алгоритм, 41, 49, 204, 329
 измерения для проведения
 сравнения, 41
 интерактивное представление, 27
 использование одного алгоритма для
 решения разных задач, 37
 история создания, 24, 45, 451
 классификация, 145
 комплексного умножения Гаусса, 464
 которые меняют мир, 451
 Краскала, 204
 Краскала для МОД, 50
 Лас-Вегаса, 375
 Лемпеля – Зива – Велча (LZW), 304
 Монте-Карло, 375
 наблюдаемые в природе, 398
 определение, 26
 оценка, 55
 поиск пути, 432
 поиска ближайшей пары точек, 373
 приготовления тостов, 26
 Прима, 203
 Прима для МОД, 50
 проектирование, 40
 рекурсивные, 27
 симплекс-метод, 4131, 418
 Флажолле – Мартина, 281
 Флойда – Уоршелла, 211, 211
 эвристические методы, 427
 элементы, 30
 Аль-Хорезми, 451
 анализ графов, 28
 анализ социальных сетей (SNA), 227
 ансамбль, 252, 314
 Аристотель, 45
 аспекты, которые могут ослабить
 положительный эффект
 параллелизма, 288
 астрономия, 261
 Белади, Ласло, 339
 Беллман, Ричард Эрнест, 348
 Беццель, Макс, 399
 бинарная куча, 154
 бинарное дерево поиска (BST), 154, 156
 бинарное дерево Хаффмана, 316
 бинарный поиск, 60, 103
 битовые потоки, 308
 Блум, Бертон Х., 275
 большие данные, 257, 261, 264
 Большой адронный коллайдер, 262
 Борувка, Отакар, 200
 Брин, Сергей, 240
 Брон, Конрад, 232
 быстрое преобразование Фурье (БПФ), 456
 Бэббидж, Чарльз, 55
 Вагнер, Роберт А., 396

- Вальд, Абрахам, 43
- веб-сети, 244
- вектор, 83
- Велч, Терри, 317
- веса, 342
- внедрение случайности, 373
- восходящий подход, 350
- временная сложность, 52
- выборка, 274, 285, 377
- вырожденная матрица, 91
- вычисление определителя, 117
- Гаусс, Карл Фридрих, 24
- Гёдель, Курт, 461
- Гемават, Санджай, 290
- генеральная совокупность, 269
- генератор истинно случайных чисел (TRNG), 493
- генератор псевдослучайных чисел, 453
- генерация псевдослучайных чисел, 29
- геномика, 262
- гидролокационные массивы, 432
- гиперкомпьютер, 463
- градиентный спуск, 404
- границы, 416
- граф пространства задачи, 51
- графический процессор (ГПУ), 31
- графы, 140
 - в числовом формате, 170
 - взвешенный, 173
 - дружбы, 227
 - ключевые атрибуты, 177
 - направленность, 141
 - невзвешенные, 201
 - неориентированный, 171
 - обход, 233
 - определение, 169
 - ориентированный, 171
 - остовное дерево, 201
 - подграфы, 230
 - построение, 141
 - ребра, 169
 - с помеченными вершинами, 173
 - случайный обход, 235
 - смешанный, 172
 - сортировка, 197
 - узлы, 169
 - центральность, 170
- данные
 - борьба с повторениями, 94
 - влияние источников, 34
 - выявление закономерностей, 457
 - использование доступных, 34
 - как найти нужные, 239
 - кэширование, 338
 - необработанные данные, 82
 - необходимость исправления, 129
 - операции с данными, 143
 - определение необходимости структуры, 126
 - поиск повсюду, 261
 - поиск с помощью словарей, 135
 - работа с деревьями, 136
 - работа с дубликатами, 129
 - работа с отсутствующими значениями, 130
 - работа с перестановками, 92
 - сжатие, 304, 305
 - скетчинг ответа на основе потоковых данных, 274
 - совместимость, 128
 - сохранение в тайне, 455
 - структурирование, 125
- Данциг, Джордж Бернард, 413
- Данциг, Тобиас, 358
- Дейкстра, Эдсгер Вибе, 211
- декартово пространство, 416
- декодирование, 319
- декораторы, 348
- деревья, 136
 - бинарные, 315
 - кучи, 139
 - несбалансированные, 139
 - остовные, 201
 - поиска, 154
 - построение, 138
 - сбалансированные, 139
 - связи, 137
 - узлы, 137
- десятичная система Дьюи, 48
- Дин, Джеффри, 290
- динамическое программирование, 349
- дифференциальная эвристика, 428

ДНК, 262, 312
 дополнение текста нулями, 325
 допустимая эвристика, 445
 допустимые ошибки, 275
 Дориго, Марко, 430
 Евклид, 24
 евклидово расстояние, 428
 единичная матрица, 90
 жадность, 330
 жадный алгоритм, 332
 жадный обмен, 335
 жадный подход, 331, 333
 задача
 3SUM, 461
 выдачи сдачи, 331
 выполнимости схемы, 404
 коммивояжера, 363
 о дробном рюкзаке, 359
 о неограниченном рюкзаке, 359
 о рюкзаке, 358
 об ограниченном рюкзаке, 359
 закон Мура, 259
 Зив, Якоб, 317
 измерение, 86
 ИИ, 254, 430
 имитация отжига, 401
 инкапсуляция, 107
 интернет вещей, 263
 калькулятор Symbolab, 119
 карты занятости, 433
 квазиполиномиальное время, 362
 Кербош, Джош, 232
 кластеризация, 182
 кластерные вычисления, 33
 клика, 230
 клоакинг, 241
 кодирование, 306
 кодирование Хаффмана, 50, 312, 317
 кодировка base64, 326
 кодировка Unicode, 307
 коллизии, 160, 161, 162
 коллинеарные точки, 461
 коммутаторы, 290
 компьютерный кеш, 338
 компьютеры, 31
 контрпримеры, 44
 корневой узел, 137
 корпус Enron, 175
 коэффициент ветвления, 53
 коэффициент заполнения, 160
 коэффициент сжатия, 311
 Краскал, Джозеф, 204
 криптография, 29, 321, 455
 криптосистема RSA, 336
 кубическая сложность, 60
 куча, 315
 куча Фибоначчи, 156
 лабиринты, 434, 444
 Лавлейс, Ада, 55
 лазерный дальномер, 432
 Лаплас, Пьер-Симон, 118
 Левенштейн, Владимир, 368
 Лемер, Деррик Генри, 27
 Лемпель, Абрахам, 317
 лидер, 432
 линейное время, 330
 линейное программирование, 412, 413
 листовой узел, 137
 логические вентили, 404
 Лойд, Сэм, 467
 локально-чувствительное хеширование
 (Locality-sensitive Hashing, LSH), 164
 локальный поиск, 372, 392
 лямбда-функция, 294
 максимальная бинарная куча, 155
 манипулирование матрицами, 117
 манхэттенское расстояние, 428
 Маркиори, Массимо, 242
 Маркхэм, Дж. Дэвид, 46
 Мартин, Найджел, 281
 Массарон, Лука, 431
 математические сопроцессоры, 31
 матрица, 83
 матрица переходов, 247
 матроиды, 334
 машина с произвольным доступом
 (Random Access Machine, RAM), 56
 машина Тьюринга, 463
 машинное обучение, 297, 431
 мемоизация, 348
 метаэвристика, 393, 460
 метеорология, 262

метод локального улучшения, 392
 метод полного перебора, 53, 359
 метод Тоома – Кука, 464
 методы объектно ориентированного
 программирования (ООП), 107
 методы преобразования Фурье, 464
 методы рандомизации, 94
 минимальная бинарная куча, 155
 минимальное остовное дерево, 201
 Митчеллом, Стюарт Энтони, 422
 многопоточность, 297
 многоядерные процессоры, 287
 Мур, Гордон, 258
 Мюллер, Джон Пол, 294, 431
 наибольший общий делитель (НОД), 25
 направленные ациклические графы
 (DAG), 197
 наследование, 107
 начальная точка, 234
 невидимый текст, 241
 Нейман, Джон фон, 461
 нейронные сети, 32
 неориентированный граф, 175
 нерешенные алгоритмические
 проблемы, 460
 нисходящий подход, 350
 Норвиг, Питер, 349
 нормальное распределение, 378
 Нотация Big O, 59
 Ньютон, Исаак, 24
 Нэш, Джон, 461
 обратная матрица, 118
 обращение матрицы, 90
 обход дерева, 138
 объем данных, 45
 односторонняя функция, 463
 океанография, 262
 определитель матрицы, 118
 оптимизация, 49, 412
 ориентированный граф, 175
 остовное дерево, 200
 Открытая инфраструктура Беркли, 34
 очередь с приоритетом, 442
 Пападимитриу, Христос, 405
 парадигмы, 329
 параллельная обработка, 79
 параллельные вычисления, 287
 паучьи ловушки, 251
 Пейдж, Ларри, 240
 перепись, 270
 переполнение ключевыми словами, 241
 Перлин, Кен, 43
 Пизанский, Леонардо (Фибоначчи), 352
 пиксель, 307
 пиррова победа, 49
 планирование, 29
 Платон, 45
 повседневные эвристики, 429
 подстановка символов, 322
 подход «разделяй и властвуй», 40
 поиск в глубину, 195, 197
 поиск в фильтре Блума, 277
 поиск в ширину, 193, 197
 поиск клика в графах, 231
 поиск кратчайшего пути, 366
 поиск пути, 432
 поисковая система, 252
 поисковые роботы, 241
 полиморфизм, 109
 потоковая обработка, 285
 поэлементное умножение, 86, 113
 преобразование, 28
 преобразование Барроуза – Уилера, 314
 преобразование Фурье, 456
 преобразовать граф в матрицу, 186
 преобразовать матрицу в одномерный
 список, 122
 приближенный поиск строк, 368
 Прикладной программный интерфейс
 (API), 178
 приоритетная очередь, 204
 проблема останова, 463
 проблема хранения данных, 132
 проблема эффективности, 55
 программное обеспечение для
 планирования потребности
 в материалах (Material Requirements
 Planning, MRP), 338
 Проксима Центавра, 261
 пропорционально-интегрально-
 дифференциальный (ПИД)
 алгоритм, 42

простая случайная выборка, 270
 пространственная сложность, 52
 пространство задачи, 52
 простые числа, 44
 процесс создания базового класса, 104
 процессоры общего назначения (ЦП),
 31
 псевдокод, 37
 псевдослучайные числа, 374
 Рабин, Майкл, 373
 равновесие Нэша, 45
 равномерное распределение, 379
 разложение Лапласа, 118
 размерность, 87
 разреженная матрица, 188
 рандомизация, 397
 рандомизация набора данных, 92
 рандомизированные алгоритмы, 373
 распараллеливание, 288
 распределение, 377
 распределенные вычисления, 34, 286
 Рассел, Стюарт, 349
 расстояние Левенштейна, 368
 расшифровка, 321
 редакционное расстояние, 465
 резервуарная выборка, 271
 результаты поискового запроса, 240
 результирующая матрица, 110
 роевой интеллект, 430
 Руа, Бернар, 221
 сжатие без потерь, 307
 сжатие данных, 304
 сжатие с потерями, 269
 Сильвер, Нейт, 270
 символьная таблица, 306
 сингулярная матрица, 91
 система управления базами данных
 (СУБД), 454
 систематическая ошибка выжившего,
 43
 скаляр, 83
 скалярное произведение, 86
 скетчинг, 281
 скрапинг, 278
 словари, 135
 случайная выборка, 93
 Смит, Адам, 332
 создание базовой матрицы, 86
 создание, чтение, обновление
 и удаление (create, read, update, and
 delete – CRUD), 143
 Соловей, Роберт М., 374
 сортировка, 28
 сортировка вставками, 147
 сортировка слиянием, 332
 социальные сети, 175
 социограммы, 227
 спамеры, 241
 списки, 165
 спутники, 26.
 срезы матрицы, 112
 стандарт шифрования Advanced
 Encryption Standard (AES), 324
 стек, 101
 стохастическое усреднение, 28.
 субоптимальное решение, 429
 суперкомпьютеры, 31, 34
 Сэки, Такакадзу, 242
 телепортация, 251
 Теллер, Эдвард, 381
 теорема Байеса, 37
 теория NP-полноты, 336
 теория игр, 466
 топологическая сортировка, 199
 топологические карты, 433
 точное среднее время между отказами,
 292
 транспонирование, 89
 Тьюринг, Алан, 463
 узлы-ветви, 137
 Уилсон, Чарльз Эрвин, 349
 умножение двух матриц, 87
 умножение Карацубы, 464
 Уоршелл, Стивен, 221
 уравнение, 25
 файловая система Google (GFS), 290
 факториал, 96
 факториальная сложность, 61
 Фано, Роберт М., 314
 Ферми, Энрико, 381
 Фибоначчи, 61
 физика, 262

фильтр Блума, 163, 275, 279
Финн, Дж., 375
Фишер, Майкл Дж., 369
Флажолет, Филипп, 281
Флойд, Роберт, 221
формула, 25
функциональные языки
 программирования, 293
хабы, 242
Хаффман, Дэвид А., 314
Хачиян, Леонид, 336
хвостовой вызов, 98
хеширование, 159
хеш-таблица, 135
хеш-функция, 160
Хоар, Тони, 151
хороший ответ на запрос, 241
Цезарь, Юлий, 455

целевая функция, 397
Шеннон, Клод, 314
шифрование, 321
Штрассен, Фолькер, 374
шум Перлина, 43
эвристика, 394
поиск с запретами, 431
эвристическая маршрутизация, 431
эвристические алгоритмы, 365
экспоненциальная сложность, 61
экспоненциальное время, 362
элементы матрицы, 111
эталонное тестирование, 57
ясновидящий алгоритм замещения, 339
ячейки кода, 76

Все права защищены. Книга или любая ее часть не может быть скопирована, воспроизведена в электронной или механической форме, в виде фотокопии, записи в память ЭВМ, репродукции или каким-либо иным способом, а также использована в любой информационной системе без получения разрешения от издателя. Копирование, воспроизведение и иное использование книги или ее части без согласия издателя является незаконным и влечет уголовную, административную и гражданскую ответственность.

Издание для досуга

для чайников

**Мюллер Джон Пол
Массарон Лука**

АЛГОРИТМЫ ДЛЯ ЧАЙНИКОВ

Главный редактор *Р. Фасхутдинов*
Руководитель направления *В. Обручев*
Ответственный редактор *Л. Салихова*
Научный редактор *А. Мелентьева*
Литературный редактор *М. Джала*
Младший редактор *М. Назаренко*
Компьютерная верстка *Т. Лукина*
Корректор *Е. Ерошкина*

Страна происхождения: Российская Федерация
Шығарушы ел: Ресей Федерациясы

ООО «Издательство «Эксмо»

123308, Россия, г. Москва, ул. Зорге, д. 1, стр. 1, эт. 20, каб. 2013. Тел.: 8 (495) 411-68-86.
Home page: www.eksmo.ru E-mail: info@eksmo.ru

Өндіруші: «Издательство «Эксмо» ЖШҚ

123308, Ресей, Мәскеу қаласы, Зорге көшесі, 1-үй, 1-құрылыс, 20 қабат, 2013-каб.
Тел.: 8 (495) 411-68-86. Home page: www.eksmo.ru E-mail: info@eksmo.ru.

Tayar berici: «Эксмо»

Интернет-магазин : www.book24.ru

Интернет-магазин : www.book24.kz

Интернет-дүкен : www.book24.kz

Импортёр в Республику Казахстан ТОО «РДЦ-Алматы».

Қазақстан Республикасына импорттаушы «РДЦ-Алматы» ЖШС.

Дистрибьютор и представитель по приему претензий на продукцию
в Республике Казахстан: ТОО «РДЦ-Алматы»

ТОО РДЦ Алматы, Алматы, ул. Домбровского, 3-а, литер Б, офис 1.

Дистрибьютор және Қазақстан Республикасында өнімге шағымдар
қабылдау жөніндегі өкіл: «РДЦ-Алматы» ЖШС.

Алматы қ., Домбровский кв., 3-а, литер Б, офис 1.

Тел.: 8 (727) 251-59-90/91/92. E-mail: RDC-Almaty@eksmo.kz

Сведения о подтверждении соответствия издания согласно законодательству РФ
о техническом регулировании можно получить на сайте Издательства «Эксмо»:
www.eksmo.ru/certification

Техникалық реттеу туралы РФ заңнамасына сай басылымның сәйкестігін растау
туралы мәліметтерді мына адрес бойынша алуға болады: <http://eksmo.ru/certification/>

Произведено в Российской Федерации
Ресей Федерациясында өндірілген

Сертификаттауға жатпайды

Дата изготовления / Подписано в печать 20.10.2025. Формат 70х100^{1/16}.

Печать офсетная. Усл. печ. л. 38,89.

Тираж экз. Заказ

ISBN 978-5-04-201470-3



9 785042 014703 >

12+

Москва. ООО «Торговый Дом «Эксмо»

Адрес: 123308, г. Москва, ул. Зорге, д. 1, строение 1.

Телефон: +7 (495) 411-50-74. **E-mail:** reception@eksmo-sale.ru

По вопросам приобретения книг «Эксмо» зарубежными оптовыми покупателями обращаться в отдел зарубежных продаж ТД «Эксмо»

E-mail: international@eksmo-sale.ru

International Sales: International wholesale customers should contact Foreign Sales Department of Trading House «Eksmo» for their orders.

international@eksmo-sale.ru

По вопросам заказа книг корпоративным клиентам, в том числе в специальном оформлении, обращаться по тел.: +7 (495) 411-68-59, доб. 2151.

E-mail: borodkin.da@eksmo.ru

Оптовая торговля бумажно-беловыми

и канцелярскими товарами для школы и офиса «Канц-Эксмо»:

Компания «Канц-Эксмо»: 142702, Московская обл., Ленинский р-н, г. Видное-2, Белокаменное ш., д. 1, а/я 5. Тел./факс: +7 (495) 745-28-87 (многоканальный).

e-mail: kanc@eksmo-sale.ru, сайт: www.kanc-eksmo.ru

Филиал «Торгового Дома «Эксмо» в Нижнем Новгороде

Адрес: 603094, г. Нижний Новгород, улица Карпинского, д. 29, бизнес-парк «Грин Плаза»

Телефон: +7 (831) 216-15-91 (92, 93, 94). **E-mail:** reception@eksmonn.ru

Филиал ООО «Издательство «Эксмо» в г. Санкт-Петербурге

Адрес: 192029, г. Санкт-Петербург, пр. Обуховской обороны, д. 84, лит. «Е»

Телефон: +7 (812) 365-46-03 / 04. **E-mail:** server@szko.ru

Филиал ООО «Издательство «Эксмо» в г. Екатеринбурге

Адрес: 620024, г. Екатеринбург, ул. Новинская, д. 2ш

Телефон: +7 (343) 272-72-01 (02/03/04/05/06/08)

Филиал ООО «Издательство «Эксмо» в г. Самаре

Адрес: 443052, г. Самара, пр-т Кирова, д. 75/1, лит. «Е»

Телефон: +7 (846) 207-55-50. **E-mail:** RDC-samara@mail.ru

Филиал ООО «Издательство «Эксмо» в г. Ростове-на-Дону

Адрес: 344023, г. Ростов-на-Дону, ул. Страны Советов, 44А

Телефон: +7(863) 303-62-10. **E-mail:** info@rnd.eksmo.ru

Филиал ООО «Издательство «Эксмо» в г. Новосибирске

Адрес: 630015, г. Новосибирск, Комбинатский пер., д. 3

Телефон: +7(383) 289-91-42. **E-mail:** eksmo-nsk@yandex.ru

Обособленное подразделение в г. Хабаровске

Фактический адрес: 680000, г. Хабаровск, ул. Фрунзе, 22, оф. 703

Почтовый адрес: 680020, г. Хабаровск, А/Я 1006

Телефон: (4212) 910-120, 910-211. **E-mail:** eksmo-khv@mail.ru

Республика Беларусь: ООО «ЭКСМО АСТ Си энд Си»

Центр оптово-розничных продаж Cash&Carry в г. Минске

Адрес: 220014, Республика Беларусь, г. Минск, проспект Жукова, 44, пом. 1-17, ТЦ «Outleto»

Телефон: +375 17 251-40-23; +375 44 581-81-92

Режим работы: с 10.00 до 22.00. **E-mail:** exmoast@yandex.by

Казахстан: «РДЦ Алматы»

Адрес: 050039, г. Алматы, ул. Домбровского, 3А

Телефон: +7 (727) 251-58-12, 251-59-90 (91,92,99). **E-mail:** RDC-Almaty@eksmo.kz

Полный ассортимент продукции ООО «Издательство «Эксмо» можно приобрести в книжных магазинах «Читай-город» и заказать в интернет-магазине: www.chitai-gorod.ru.

Телефон единой справочной службы: 8 (800) 444-8-444. Звонок по России бесплатный.

Интернет-магазин ООО «Издательство «Эксмо»

www.eksmo.ru

Розничная продажа книг с доставкой по всему миру.

Тел.: +7 (495) 745-89-14. **E-mail:** imarket@eksmo-sale.ru



ЧИТАЙТЕ
И СЛУШАЙТЕ
в Литрес

ЧИТАЙ-ГОРОД

БОМБОРА
ИЗДАТЕЛЬСТВО

БОМБОРА – лидер на рынке полезных и вдохновляющих книг. Мы любим книги и создаем их, чтобы вы могли творить, открывать мир, пробовать новое, расти. Быть счастливыми. Быть на волне.

@ bombora.ru | bomborabooks | bombora



Хочешь стать
автором «Эксмо»?



ТЕРИТОРИЯ
КНИЖНЫЙ МАГАЗИН
Официальная франшиза
издательства «Эксмо»



eksmo.ru
Официальный
интернет-магазин
издательства «Эксмо»

Секретное оружие для эффективного использования алгоритмов

От ленты новостей до расчета страховых взносов — алгоритмы пронизывают практически все сферы нашей жизни, формируя современное общество и личную реальность. На первый взгляд они могут показаться непостижимо сложными, но на самом деле каждый способен освоить — и творчески применять — эти мощные инструменты для решения любых задач!

Книга «Алгоритмы для чайников» расскажет, что такое алгоритмы, как они функционируют, где встречаются (спойлер: повсюду!), кто стоит за созданием ключевых современных алгоритмов (это удивительно, но их корни уходят к древнегреческому философу!) и как вы можете начать создавать их самостоятельно.



В этой книге вы найдете:

- **Наглядные материалы:** десятки графиков и диаграмм, помогающих интуитивно понять внутреннюю логику алгоритмов
- **Практический код:** ссылки на онлайн-репозиторий GitHub для доступа к актуальным примерам кода
- **Легкий старт в программировании:** пошаговые инструкции по работе с Google Colaboratory — удобной средой программирования, не требующей установки и запускающейся прямо в браузере

Джон Пол МЮЛЛЕР — автор и технический редактор. Он написал 121 книгу и более 600 статей.

Лука МАССАРОН — специалист в обработке данных и директор по маркетинговым исследованиям. Также является экспертом по машинному обучению программы Google Developer Expert (GDE).

Все издания серии
для
Чайников —
это ваш надежный проводник
в мире новых знаний!

Книга дает хорошую фундаментальную базу, учит находить решения сложных задач и видеть алгоритмы в повседневной жизни. Такие знания помогают замечать системы и закономерности, на которых строятся современные технологии. Авторы на простых примерах показывают, что за каждым действием — от древних вычислений до работы нейросетей — стоит понятная логика, доступная каждому.

Андрей Иванов,
методолог, проджект-раннер
и основатель Fractales Agency

ISBN 978-5-04-201470-3



9 785042 014703 >

БОМБОРА
ИЗДАТЕЛЬСТВО

БОМБОРА — лидер на рынке полезных и вдохновляющих книг. Мы любим книги и создаем их, чтобы вы могли творить, открывать мир, пробовать новое, расти. Быть счастливыми. Быть на волне.